Universidad Zaragoza

Trabajo Fin de Grado

Detección de reuso en LLC mediante filtros Bloom.

Reuse detection in LLC through Bloom filters.

Autor

Noelia Oliete Escuín

Director

Pablo Enrique Ibáñez Marín

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2022

Detección de reuso en LLC mediante filtros Bloom

RESUMEN

El acceso a memoria principal ha sido siempre el cuello de botella más notable en un sistema. Es por ello, por lo que establecer un uso eficiente del espacio disponible en cada nivel de la jerarquía de memoria permitirá mejorar notablemente las prestaciones del sistema. En este proyecto se estudia el uso de filtros de reuso para paliar dicha problemática. Estas estructuras permiten filtrar el contenido introducido en la cache de último nivel (LLC) para conseguir almacenar solamente aquellos bloques de memoria que presentan una alta probabilidad de reuso. Para ello, deben recordar direcciones de bloque que han provocado fallo en la LLC recientemente. Seshadri et al. (2012) proponen por primera vez el uso de esta estructura implementada mediante un filtro Bloom. Más tarde, se presentan otro tipo de implementaciones como ReD, donde se utiliza una estructura de datos organizada de forma similar a una cache.

El filtro Bloom es una estructura probabilística que permite determinar si un elemento pertenece o no a un conjunto. Inicialmente fue diseñado para conjuntos estáticos, pero con el paso del tiempo se han llevado a cabo múltiples modificaciones a la estructura con el fin de adaptarla a nuevos escenarios donde se necesita trabajar con conjuntos dinámicos. Esto ha dado lugar a nuevos tipos de filtros Bloom con diferentes funcionalidades y características configurables. En este proyecto, se han seleccionado e implementado aquellos filtros Bloom que mejor se adaptan al problema. Estos son: Básico extendido, A2, SBF y SetReset. También se ha implementado un filtro ideal, *Brain*, usado para calificar la precisión de recuerdo de cada uno de los filtros estudiados. Los filtros se han implementado en el simulador *ChampSim* sobre sistemas mono-core y multi-core, y se han sintetizado sobre una FPGA. De esta forma, se busca conocer el número óptimo de direcciones a recordar, para que el sistema presente las mejores prestaciones posibles, y ajustar los parámetros de cada filtro Bloom en base a las métricas precisión de recuerdo, instrucciones por ciclo (IPC) y fallos por cada mil instrucciones (MPKI). Adicionalmente, se comparan las prestaciones de todos los filtros de reuso estudiados y el coste hardware de los filtros Bloom seleccionados.

En base a los resultados obtenidos se concluye que no existe una correspondencia exacta entre tener un precisión de recuerdo alta y tener mejores prestaciones (IPC y MPKI). Por otro lado, utilizar un filtro de reuso permite mejorar las prestaciones tanto del sistema mono-core como del multi-core. Para el sistema mono-core se consiguen mejores resultados con la estructura ReD, mientras que para el multi-core es preferible incorporar el filtro A2 o Básico extendido. Por último, el impacto en área obtenido para los filtros Básico extendido, A2 y SetReset es similar e inferior al de SBF.

Índice

1. Introducción **1**

1.1. Motivación . . . 1

1.2. Objetivos y Tareas . . . 2

1.3. Descripción del contenido . . . 3

2. Descripción del problema **5**

2.1. Selección de contenido mediante filtro de reuso . . . 6

3. Detector de reuso mediante filtro de reuso **9**

3.1. *Brain* . . . 9

3.2. ReD . . . 10

3.3. Filtro Bloom . . . 11

3.4. Básico extendido . . . 12

3.5. A^2 . . . 13

3.6. Stable Bloom Filter, SBF . . . 14

3.7. SetReset . . . 14

4. Metodología **16**

4.1. Simulador *ChampSim* . . . 16

4.2. Carga de trabajo . . . 17

4.3. Herramienta de síntesis . . . 18

4.4. Casos de prueba . . . 19

4.5. Métricas . . . 19

5. Resultados **20**

5.1. Configuración de *Brain*: estudio de tamaño óptimo . . . 21

5.2. Ajuste de filtros Bloom . . . 22

5.2.1. Básico extendido . . . 23

5.2.2. A2 . . . 25

5.2.3. SBF . . . 28

5.2.4. SetReset 31
5.3. Comparación global de prestaciones 33
5.3.1. Función hash alternativa 34
5.4. Coste en área 35
5.5. Multi-core 38

6. Conclusiones **40**

Bibliografía **42**

Lista de Figuras **45**

Lista de Tablas **47**

Anexos **48**

A. Algoritmos de reemplazo: LRU y SRRIP **49**

B. Funciones hash **50**
B.1. MD5 50
B.2. H3 53

C. Pruebas de caja negra: Tablas de decisión **54**

D. SetReset: Subconjuntos **59**

Capítulo 1

Introducción

1.1. Motivación

El acceso a memoria principal ha sido siempre el cuello de botella más notable en un sistema. En los últimos años han aparecido nuevas aplicaciones que requieren tanto de un gran ancho de banda de memoria, como de una latencia reducida. Ambos requisitos incrementan la presión ejercida sobre la memoria, por lo que establecer un uso eficiente del espacio disponible en cada nivel de la jerarquía permitirá mejorar notablemente las prestaciones del sistema.

Idealmente, para conseguir un buen uso del espacio, la jerarquía de caches del sistema solamente debería almacenar aquellos bloques que van a ser utilizados en un futuro cercano. Entre las técnicas utilizadas para conseguir dicho comportamiento se encuentran los algoritmos de reemplazo. Estos tratan de expulsar de la cache aquellos bloques que no van a ser utilizados en el futuro. Se sabe que existe una gran correlación entre el uso reciente de un bloque y su posible uso futuro. Por ello, uno de los algoritmos de reemplazo más usados consiste en expulsar el bloque menos recientemente utilizado (LRU por sus siglas en inglés). Por otro lado, otra de las técnicas es la prebúsqueda. Esta aprovecha patrones recurrentes de acceso a direcciones de memoria, para adivinar los accesos futuros y cargar los bloques con antelación. Por último, uno de los focos de investigación más recientes busca crear estructuras, llamadas filtros de reuso, que permitan filtrar el contenido introducido en la cache de último nivel (LLC), con el fin de almacenar solamente aquellos bloques con una alta probabilidad de uso futuro. Se asume que un bloque que ha sido solicitado recientemente más de una vez a memoria principal tiene mayor probabilidad de volver a ser utilizado (reuso). Por ello, los filtros de reuso recuerdan el conjunto de las direcciones de bloque que han provocado fallo en la LLC recientemente y que por tanto, se han cargado desde memoria principal. De esta forma, clasifican estas mismas direcciones en reusadas o no, comparándolas con aquellas que se encuentran almacenadas en la estructura. Sólo los bloques clasificados

como reusados se guardan en la LLC. Actualmente, Intel ya usa filtros de reuso en la cache de último nivel de sus procesadores a partir de Xeon Scalable (SkyLake) [1].

Como ejemplos de este tipo de estructuras se encuentran ReD [2] y el filtro Bloom [3], ambas estudiadas en este proyecto. ReD está conformada por vías y conjuntos, tal y como ocurre en una cache. En cada vía almacena una etiqueta comprimida, creada a partir de una dirección de bloque, con la que se realiza la detección de reuso. Por otro lado, la principal estructura analizada en este proyecto, el filtro Bloom, es una estructura probabilística que permite determinar si un elemento pertenece o no a un conjunto. En el marco de trabajo de este proyecto, los elementos son los bloques provenientes de memoria principal hacia la jerarquía de caches y el conjunto es el de bloques recientemente utilizados. No obstante, esta estructura presenta un gran problema al no incorporar una funcionalidad de borrado que le permita olvidar los elementos que pertenecen al conjunto, lo cual hace que la clasificación del filtro sea incorrecta puesto que todos los bloques serán clasificados como con reuso.

1.2. Objetivos y Tareas

El objetivo de este trabajo es estudiar diferentes implementaciones de filtros Bloom que cuentan con una adecuada funcionalidad de borrado, así como analizar la precisión que ofrecen para recordar bloques recientemente utilizados y las prestaciones que obtienen al ser incorporados entre la LLC y la memoria principal, en sistemas mono-core y multi-core. Asimismo, se desea realizar una comparativa de prestaciones con la estructura ReD y conocer el impacto en área de cada uno de los filtros Bloom estudiados.

Como primer paso del trabajo, se ha realizado una tarea de documentación acerca de la organización de ReD y del filtro Bloom, para comprender la funcionalidad de las estructuras y sus distintas aplicaciones. Seguidamente, se han seleccionado los distintos diseños de filtros Bloom que se adecuan a las necesidades del problema. Estos son el filtro Básico extendido [3], el filtro A^2 [4], el filtro Stable Bloom Filter [5] y el filtro SetReset, que es una nueva propuesta presentada en este proyecto. Una vez escogidos los filtros, el siguiente paso ha sido la implementación y verificación de cada filtro en el lenguaje de programación *C++* y en el de descripción hardware *VHDL*. Posteriormente se han incorporado al simulador *ChampSim* [6] los filtros Bloom y ReD. *Champsim* es un entorno suministrado en distintas competiciones internacionales sobre reemplazo, prebúsqueda y predicción de saltos, y es ampliamente utilizado en trabajos de investigación sobre arquitectura de computadores. A continuación, se han lanzado simulaciones variando las características de cada filtro, como el tamaño, utilizando las

trazas SPEC CPU2006 [7] y se han sintetizado los 4 diseños de filtro Bloom estudiados sobre una FPGA, con el fin de conocer su impacto en área. Por último, se han analizado los resultados obtenidos y se ha procedido a la redacción de esta memoria. En la Figura 1.1 se puede observar la distribución aproximada del tiempo dedicado a cada tarea del proyecto, donde el total de horas es de 360. El entorno de simulación *ChampSim* se había estudiado y configurado previamente durante la realización de unas prácticas con el grupo de investigación de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ). Esta tarea supuso 100 horas adicionales de trabajo que no han sido contabilizadas en el diagrama.

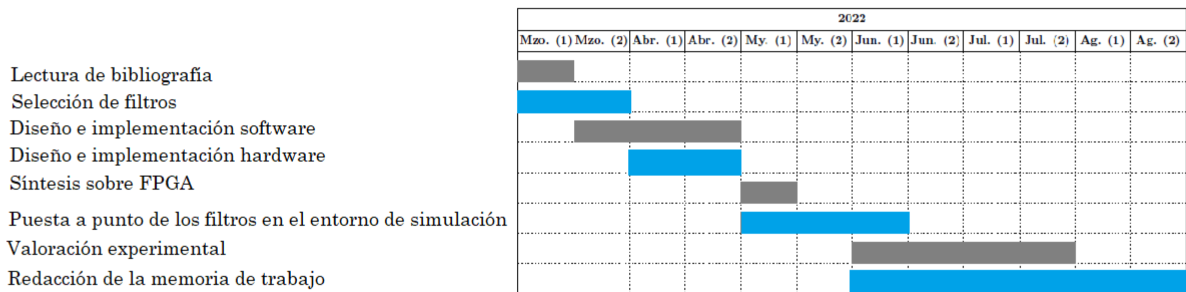

Figura 1.1: Diagrama de Gantt. Tiempo dedicado al proyecto por tareas.

1.3. Descripción del contenido

La memoria del proyecto consta de seis capítulos incluyendo la presente introducción. En el capítulo 2 se lleva a cabo una contextualización de la problemática que se trata de resolver en este trabajo mediante el uso de filtros Bloom, junto con una descripción de otras alternativas creadas con el mismo fin. En el siguiente capítulo, el capítulo 3, se realiza una descripción de la composición y funcionalidad de las tres estructuras estudiadas en este proyecto. La primera estructura es *Brain*, que es una estructura ideal, capaz de recordar exactamente un número de direcciones de bloque estipulado, y cuya finalidad es contrastar la capacidad de recuerdo de los distintos filtros estudiados. La segunda estructura es ReD y la última, el filtro Bloom. Además, también se detallan estos aspectos para las distintas implementaciones seleccionadas; Básico extendido, SBF, A2 y SetReset.

En el capítulo 4 se explica la metodología empleada para el trabajo realizado; entorno de simulación y trazas, herramienta de síntesis, diseño de casos de prueba y métricas (PR, IPC y MPKI).

En el capítulo 5 se exponen y analizan los resultados obtenidos en los distintos experimentos llevados a cabo. En el primer experimento se realiza la configuración de recuerdo de la estructura *Brain*, para seleccionar el marco temporal de recuerdo óptimo.

Después, se ajustan los parámetros de cada filtro Bloom en base a la precisión de recuerdo, PR, al número de fallos en LLC por cada mil instrucciones ejecutadas, MPKI, y al número de instrucciones ejecutadas por ciclo, IPC. Seguidamente, se comparan las prestaciones obtenidas con ReD, *Brain* configurado con la capacidad de recuerdo óptima y las mejores configuraciones de los filtros Bloom en un sistema mono-core. A continuación, se analiza el impacto en área de cada uno de los filtros Bloom y finalmente, se repite la comparativa de prestaciones (MPKI e IPC) de los filtros estudiados para un sistema multi-core.

En el capítulo 6, se extraen las conclusiones del trabajo.

Capítulo 2

Descripción del problema

Entre los caminos más críticos presentes en un sistema se encuentra el acceso a memoria principal, ya que obtener la información que esta almacena requiere de una gran latencia. Por este motivo, los sistemas actuales cuentan con una jerarquía de caches que permite disminuir la latencia media de acceso, y por ende, aumentar las prestaciones del sistema a través del uso de la localidad temporal y espacial de las posiciones de memoria solicitadas. Cuando un programa accede a un elemento, existe una alta probabilidad de que ese mismo elemento vuelva a ser accedido en un futuro cercano. Esto es lo que denotamos como localidad temporal. En cuanto a la localidad espacial, esta se define como la tendencia de un sistema a referenciar posiciones de memoria contiguas a la solicitada. Una jerarquía de caches está conformada por dos o más niveles de caches con diferentes tamaños y latencias, que aumentan con cada nivel, es decir, la cache de primer nivel tiene un tamaño y una latencia menor que la de segundo nivel y así, con todos los niveles de la jerarquía. Típicamente, está conformada por tres niveles denotados como L1, L2 y LLC, siendo L1 el primer nivel, L2 el segundo nivel y LLC, la cache de último nivel. Además, la jerarquía de caches cuenta con una política de contenido que define la gestión que se realiza sobre la información almacenada en cada uno de los niveles, con el fin de aprovechar de forma eficiente el espacio disponible. Aunque existen una gran variedad de políticas, todas ellas surgen de combinar la política de inclusión y exclusión en cada nivel. Decimos que la jerarquía presenta una política de inclusión si la cache de último nivel almacena todos los bloques que están presentes en la de nivel inferior. En cambio, si ninguno de los bloques almacenados en cada cache coincide, la jerarquía presenta una política de exclusión.

Las caches que forman la jerarquía, independientemente del nivel al que pertenezcan, presentan la misma estructura. Dicha estructura, está conformada por uno o más conjuntos que son divididos a su vez en vías, donde se encuentran almacenados los bloques de memoria. A través del tamaño de estos bloques, se busca aprovechar la

localidad espacial. Esto es, porque si se almacena un bloque con un tamaño superior al requerido en la petición a memoria, se guarda, no solo la información solicitada, si no también la contigua a esta. De esta forma, si una vez realizada la petición inicial a memoria que provoca el almacenamiento del bloque, se solicita una dirección contigua a esta, podrá obtener la respuesta con la latencia de la cache donde se encuentra almacenado el bloque, en vez de pagar el coste de solicitarlo nuevamente a una cache de nivel superior o a memoria principal. Por otro lado, cada una de las caches cuenta con un algoritmo de reemplazo. Este algoritmo se encarga de decidir la vía o bloque a expulsar, denotado como bloque víctima, cuando se inserta un nuevo bloque en el conjunto y este se encuentra completo. Existen múltiples algoritmos de reemplazo, donde muchos de ellos buscan aprovechar la localidad temporal. Para ello, escogen como bloque víctima aquel ha pasado mas tiempo sin ser solicitado, entre todos los bloques del conjunto donde se inserta el bloque. Least Recently Used, LRU, y Static Re-Reference Interval Prediction, SRRIP, son los algoritmos utilizados en el presente trabajo y cuyo funcionamiento se detalla en el Anexo A.

2.1. Selección de contenido mediante filtro de reuso

Muchos de los bloques solicitados, y posteriormente almacenados en la jerarquía de caches, permanecen almacenados en los niveles inferiores mientras el sistema los utiliza. Sin embargo, una vez expulsados de estos niveles, nunca vuelven a ser solicitados a la LLC. Se denota a este tipo de bloques como no reusados en la LLC, ya que no son pedidos más de una vez en un determinado marco temporal. El problema que crean este tipo de bloques es que provocan la expulsión de otros que si presentan reuso. Lo que se traduce en una bajada de prestaciones del sistema, debida al aumento del número de accesos a memoria principal para solicitar nuevamente los bloques con reuso que fueron expulsados.

Todo ello da lugar a una optimización que consiste en almacenar, en la cache de último nivel, los bloques que no presentan reuso con la prioridad de expulsión más alta o, directamente, realizar *bypass*. Se denota *bypass* al proceso por el cual el bloque solicitado a memoria principal no es almacenado en la LLC, pero si se entrega a las caches privadas. Para ello, es necesario un mecanismo que permita detectar si los bloques solicitados a memoria principal, debido a un fallo en LLC, presentan o no reuso. A este mecanismo se le denota como filtro de reuso, y ya se encuentra incorporado en el último nivel de cache de los procesadores de Intel a partir de Xeon Scalable (SkyLake) [1].

El filtro de reuso es una estructura que permite almacenar un conjunto de elementos.

En este caso, el conjunto almacenado es una ventana temporal de las direcciones de bloque solicitadas a memoria principal, para conocer si presentan o no reuso. Cuando las caches de nivel inferior realizan una petición a la LLC, y ésta no contiene el bloque solicitado, se obtiene el bloque de memoria principal. Es en este momento cuando se consulta el filtro. Si la dirección del bloque solicitado se encuentra almacenada en el filtro, significa que el bloque ha sido utilizado recientemente y por tanto, se guarda en LLC. Por el contrario, si la dirección del bloque solicitado no está en el filtro, el bloque se entrega a las caches de nivel inferior, pero no se almacena en la LLC. Además, en este último caso, la dirección se guarda en el filtro para detectar posible reuso en el futuro. La figura 2.1 muestra gráficamente este comportamiento.

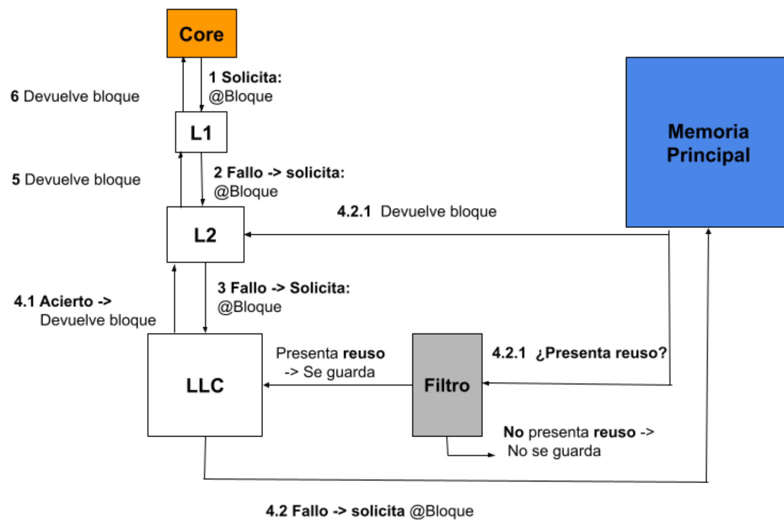

Figura 2.1: Representación gráfica del funcionamiento del filtro de reuso en la jerarquía de memoria

Seshadri et al. (2012) [8] proponen por primera vez el uso de un filtro de reuso, implementado mediante un filtro Bloom, para asignar distintas prioridades de reemplazo a los bloques según tengan o no reuso. Más tarde se propone usar el filtro para decidir si el bloque se guarda en cache o no, y se implementa mediante una estructura de datos organizada de forma similar a una cache, llamada ReD [2].

En este trabajo revisitamos la implementación mediante filtros Bloom. A priori, esta estructura es más simple y puede ser más eficiente que la usada por ReD. Sin embargo, presenta un gran problema a la hora de limitar la ventana temporal de recuerdo. En otras palabras, olvidar aquellos bloques que ya no pertenecen a dicha ventana temporal. Se han propuesto varias mejoras para solucionar esta problemática, como el borrado completo de la ventana temporal recordada. En el capítulo 3 se presentan algunas de ellas y se seleccionan las que mejor se adaptan al problema que tratamos de resolver. Después, en el capítulo 5, se dimensionan correctamente y se comparan entre si y con

la implementación ReD.

Capítulo 3

Detector de reuso mediante filtro de reuso

En este capítulo se presentan y describe el funcionamiento de los tres tipos de filtros de reuso estudiados. Estos son *Brain*, ReD y el filtro Bloom junto con distintas variaciones del mismo.

3.1. *Brain*

Con el propósito de calificar la precisión con la que cada uno de los filtros Bloom estudiados son capaces de recordar las direcciones de los bloques analizados se ha diseñado una estructura nombrada *Brain*. *Brain* es una estructura ideal, capaz de recordar los últimos b bloques, en este caso, solicitados a memoria principal por la cache de último nivel. Asimismo, esta información puede ser consultada con el fin de conocer si un bloque pertenece o no al conjunto de los b más recientes.

Por lo tanto, comparando la información obtenida tras consultar cada uno de los filtros estudiados con la obtenida con *Brain*, se puede conocer si la respuesta del filtro es o no acertada. En un caso ideal, el filtro debería generar las mismas respuestas que *Brain*, pero al tratarse de una estructura probabilística pueden o no coincidir en sus respuestas dando lugar a la aparición de las métricas verdadero positivo, verdadero negativo, falso positivo y falso negativo. Cuando el filtro Bloom clasifica el bloque consultado como reusado, diremos que la consulta es un verdadero positivo si la estructura *Brain* determina que se encuentra entre los b bloques más recientemente usados, y diremos que es un falso positivo en caso contrario. Por otro lado, cuando el filtro clasifica un bloque como sin reuso, diremos que la consulta es un verdadero negativo si *Brain* tampoco recuerda dicho bloque entre los más recientes, mientras que en caso contrario diremos que es un falso negativo.

La estructura *Brain* está conformada por una matriz de tamaño NxM, siendo N y M números naturales que representan el número de filas y el número de columnas

respectivamente. Cada columna almacena información sobre un bloque, su dirección y su estampa temporal, la cual representa el instante temporal en el que ha sido insertado o consultado. El valor de la estampa temporal viene determinado por una estampa temporal global t, representada mediante un número natural, que aumenta su valor en uno con cada acceso a la estructura. Además, para indexar las N filas de la matriz, se incluye una función hash que transforma la dirección del bloque consultado, en un valor entre 0 y $N-1$ con el que se realiza el acceso.

Las dos funcionalidades que presenta la estructura *Brain* son: consultar si un bloque pertenece a los b más recientes e insertar una dirección de bloque. Al realizar una operación de consulta, primeramente se actualiza la estampa temporal global sumando uno a su valor, $t+1$. Después, se obtiene la fila en la que debería estar almacenado. Seguidamente, se realiza una búsqueda entre las M columnas de la fila indexada, comenzando desde la última que contiene un elemento almacenado. En esta búsqueda se compara la dirección del bloque consultado, con la almacenada en cada una de las columnas. Si no se encuentra, se añade la dirección del bloque junto con la estampa temporal actual t, a la primera columna libre, y devuelve "no es reciente". Si la encuentra, se comprueba si la diferencia entre t y la estampa temporal almacenada es menor a b. Si se cumple, la respuesta devuelta es "reciente" y en caso contrario, la respuesta es "no es reciente" y se actualiza su estampa temporal al valor t.

3.2. ReD

ReD [2] es un filtro de reuso utilizado en el presente trabajo para contrastar el rendimiento obtenido con cada uno de los filtros Bloom. Su estructura está formada por conjuntos y vías, tal y como la de una cache, donde el algoritmo de reemplazo utilizado es FIFO. Este algoritmo elige como bloque víctima aquel que fue insertado con anterioridad. Cada vía almacena una etiqueta comprimida por dirección de sector, en vez de por dirección de bloque, entendiendo por sector a un conjunto de bloques de memoria consecutivos y alineados al tamaño de sector, además de un bit de presencia por bloque. Asimismo, cada conjunto cuenta con $log_2(Número\ de\ vías)$ bits para el funcionamiento del algoritmo de reemplazo y un bit de validez por vía. Este bit permite conocer si la vía se encuentra ocupada.

La configuración ideal de ReD ocupa un total de 28 KB y tiene una cobertura de expulsión de 2 MB. Se define cobertura de expulsión como el espacio de memoria ocupado por los bloques expulsados, para los que el detector de reuso puede hacer seguimiento en un determinado momento. Está formada por 1024 conjuntos de asociatividad 16, un tamaño de sector de 2 bloques y utiliza etiquetas de 10 bits.

Por lo tanto, cada entrada del detector de reuso requiere 14 bits (10 de etiqueta, 2 de presencia de bloque, 1 de reemplazo y 1 de validez).

ReD declara que un bloque presenta reuso si su dirección se encuentra almacenada en su estructura. Para ello, compara la etiqueta del bloque analizado, que es construida a partir de su dirección, con las etiquetas de las vías del conjunto asociado. En caso contrario, declara que el bloque no presenta reuso y lo recuerda. De esta forma, si vuelve a ser consultado antes de que pueda ser expulsado, presentará reuso y ReD lo detectará. Como excepción, bloques que no presenten reuso se insertan en la cache con una probabilidad de 1/32.

3.3. Filtro Bloom

El filtro Bloom [3] es una estructura probabilística, creada por Burton Howard Bloom en 1970, que permite determinar si un elemento pertenece o no a un conjunto.

Se compone de una tabla formada por celdas de un bit y una o varias funciones hash. Cada elemento consultado se identifica con una clave con la que cada función hash genera un índice para acceder a la tabla. De esta forma, se puede consultar su contenido o escribir sobre ella. Las funciones hash estudiadas en este proyecto son MD5 [9] y H3 [10], cuyo funcionamiento puede verse detallado en el Anexo B. Añadir un elemento al conjunto consiste en escribir un 1 en todas las celdas asociadas a su clave. Para consultar si un elemento pertenece al conjunto se leen todas las celdas asociadas a su clave. El elemento es clasificado como "pertenece al conjunto", si el valor de todas las celdas es 1. En caso contrario, es clasificado como "no pertenece al conjunto".

Inicialmente, el filtro Bloom fue diseñado para trabajar con conjuntos estáticos. De esta forma, todos los miembros del grupo ya se encuentran definidos dentro de la propia estructura. Por este motivo, la única funcionalidad que presenta es la de consultar si un elemento pertenece o no al conjunto. Al tratarse de un conjunto estático, la estructura puede presentar falsos positivos, pero nunca falsos negativos. Se considera un falso positivo un elemento que no pertenece al conjunto pero que el filtro determina que si pertenece, y un falso negativo un elemento que pertenece al conjunto pero que el filtro determina lo contrario.

Una de las aplicaciones de esta estructura fue ayudar a mantener la coherencia de los bloques de la jerarquía de caches [11]. Pero para ello, se necesitaba trabajar con un conjunto dinámico, así que se incorporó al filtro Bloom la posibilidad de añadir y borrar elementos.

A lo largo de los años se han creado múltiples alternativas respecto al diseño del

filtro Bloom original. La primera tarea de este proyecto consistió en analizar todas las propuestas de filtros Bloom para seleccionar aquellas que se adaptan mejor al escenario del proyecto. En nuestro caso, un elemento se refiere a un bloque de memoria, identificado por su dirección, y el conjunto es el de los bloques que han sido pedidos a memoria principal en un pasado reciente. Por tanto, el filtro ha de tener funcionalidad para añadir elementos y para borrarlos. Este motivo hace que todos aquellos filtros que no incluyan estas funcionalidades queden descartados, como el presentado por Kiss et al. [12]. También se descartan todos aquellos filtros que no se adaptan a la problemática del proyecto. Estos son aquellos cuya estructura es descentralizada [13], asocia un par clave-valor a cada elemento insertado [14], determinar si un elemento pertenece a uno más conjuntos [15] o está conformada por capas de filtros [16]. El último tipo de filtro descartado es el filtro atenuado [17], ya que la consulta al filtro se realiza mediante un algoritmo de profundidad, el cual requiere un gran coste temporal. Finalmente, los filtros que han sido escogidos son el filtro Básico extendido [3], A^2 [4] y Stable Bloom Filter [5], los cuales fueron diseñados para recordar los elementos más recientes en entornos de *streaming data*. También se estudia en este proyecto una nueva propuesta de filtro Bloom llamado SetReset.

3.4. Básico extendido

A medida que se insertan nuevos elementos en el filtro, la cantidad de celdas con valor 1 es cada vez mayor, provocando el aumento de falsos positivos. El filtro Bloom Básico extendido [3] surge con el propósito de solucionar este problema incorporando una funcionalidad de borrado al filtro Bloom base.

Esta extensión se conforma por los mismos elementos que el filtro original, es decir, por una tabla formada por celdas de un bit y una o más funciones hash. Estas funciones se encargan de transformar las direcciones de bloque en los índices utilizados para indexar la tabla, con el fin de consultar o escribir su contenido. Por otro lado, las funcionalidades con las que cuenta son las ya existentes en el filtro base, insertar un elemento y consultar si un elemento presenta o no reuso, más una de borrado. La inserción y consulta tienen el mismo comportamiento que el filtro base. En cuanto a la funcionalidad de borrado, esta es aplicada cuando el filtro almacena una cantidad estipulada de información, en otras palabras, cuando la tabla del filtro tiene un determinado número de celdas con el valor 1, denotado en este proyecto como nivel máximo de llenado. Esto se traduce en un borrado completo de la información, es decir, todas las celdas de la tabla se rellenan con el valor 0. En consecuencia, la operación de borrado implica la pérdida de todo el conocimiento adquirido por la estructura y

la aparición de falsos negativos, que son elementos que pertenecen al conjunto pero el filtro declara que no pertenecen.

3.5. A^2

El filtro Bloom A^2 [4] es una mejora del filtro Double Buffering [18]. Double Buffering propone el uso de dos tablas en vez de una, tal y como ocurre en el original. Ambas tablas tienen el mismo tamaño, están formadas por celdas de un bit e indexadas mediante las mismas funciones hash. Además, estas se encuentran activas de manera alternativa, es decir, una guarda información mientras que la otra se encuentra totalmente vacía. Por el contrario, A^2 aprovecha la estructura al completo manteniendo ambas tablas activas, denotadas como *active1* y *active2*. La tabla *active1* almacena la información más reciente, mientras que *active2* se encarga de la información más antigua. En otras palabras, *active1* guarda la información de bloques con reuso recientemente utilizados y *active2*, de bloques con reuso que fueron utilizados en un marco temporal anterior.

Cuenta con tres funcionalidades; insertar un elemento, consultar si un elemento presenta o no reuso y borrado. Cuando se realiza una consulta sobre el filtro, primeramente se consulta la tabla *active1*. Si el valor de todas las celdas indexadas a través de los índices obtenidos por las hashes es 1, el bloque presenta reuso. Si no es así, se lleva acabo la misma consulta en la tabla *active2* cuya respuesta, obtenida de la misma forma que la anterior, será el resultado final de la consulta. Seguidamente, se realiza una inserción sobre las celdas de la tabla *active1*. En cuanto a la funcionalidad de borrado, esta se activa cuando la tabla *active1* llega a un nivel de llenado determinado. En este momento se pasa el contenido de *active1* a *active2* y se borra el contenido de *active1*. Esto no implica la total perdida de información puesto que aquella que ha sido adquirida por *active2* puede ser igualmente consultada.

Utilizando los mismos recursos que el filtro Bloom original, es decir, las mismas funciones hash y el mismo tamaño de tabla, A^2 es capaz de trabajar con conjuntos dinámicos. Esto es debido a que incorpora una funcionalidad de borrado la cual provee una mejora en la detección de reuso, respecto a la proporcionada en el filtro Básico extendido, dado que solo implica la perdida parcial de la información almacenada. Además, mantiene durante más tiempo en el conjunto aquellos bloques con reuso utilizados con mayor frecuencia.

3.6. Stable Bloom Filter, SBF

En el filtro Bloom original no existe una forma de distinguir los elementos recientes de los antiguos, ya que no se almacena información temporal. El filtro SBF [5] nace con el propósito de diseñar un filtro Bloom que permita tener este tipo de comportamiento.

Está formado por una tabla compuesta por celdas de uno o más bits, d, donde el valor mínimo de cada celda es 0 y el valor máximo es 2^{d-1} y una o más funciones hash, que generan los índices con los que se indexa dicha tabla a partir de las direcciones de bloque.

Cuenta con tres funcionalidades; insertar un elemento, consultar si un elemento presenta o no reuso y borrado. Cuando se consulta un bloque, el filtro determina que presenta reuso si todos los valores de las celdas, indexadas por los índices obtenidos paralelamente a través de las funciones hash, tienen un valor mayor a 0. En caso contrario, determina que no presenta reuso y ejecuta la funcionalidad de borrado. El borrado escoge aleatoriamente un número de celdas y les disminuye su valor en uno. Por último, se escribe el máximo valor posible por celda, 2^{d-1}, en las celdas indexadas por los índices obtenidos mediante las funciones hash. De esta forma, estas celdas no se verán afectadas por la funcionalidad de borrado en caso de haber sido seleccionadas aleatoriamente en el paso previo. Además, se mantendrán activas, es decir, con un valor distinto de cero, durante más tiempo.

Respecto al filtro Bloom original, SBF necesita más recursos ya que su tabla cuenta con un tamaño igual o superior a la del original. Esto es porque sus celdas pueden tener un tamaño mayor a un bit. Además, para implementar la funcionalidad de borrado es necesario un generador de números aleatorios y un restador por cada celda borrada. A pesar de su coste, SBF consigue mantener durante más tiempo aquellos bloques que son utilizados con más frecuencia. Esto se debe a que cuando se consulta un elemento, el valor de sus celdas toma el máximo valor posible. Por tanto, si este elemento es consultado con gran frecuencia, sus celdas siempre se mantendrán en un valor distinto de 0, a pesar de que puedan ser seleccionadas aleatoriamente al ejecutar la funcionalidad de borrado.

3.7. SetReset

Una nueva propuesta de filtro Bloom presentada en este proyecto es el filtro SetReset. El nombre asignado al filtro se debe a que separa el conjunto de elementos a almacenar en dos subconjuntos llamados *Set* y *Reset*. Las direcciones de bloque pertenecientes al conjunto *Set*, se recuerdan escribiendo un 1 en las celdas asociadas,

mientras que las del conjunto *Reset*, se recuerdan escribiendo un 0. Para conseguir un buen rendimiento en el sistema, el número de bloques pertenecientes a cada subconjunto debe ser equitativo.

Se conforma de una tabla de celdas de un bit, cuyo valor inicial se escoge de forma aleatoria, y un conjunto par de funciones hash, ya que cada subconjunto utiliza las suyas propias.

Las funcionalidades con las que cuenta son; insertar un elemento y consultar si un elemento pertenece o no al conjunto de bloques reusados. Al realizar una consulta en el filtro, primeramente se obtiene el valor de las celdas indexadas por los índices obtenidos a través de las funciones hash. La respuesta a la consulta es "presenta reuso", si el valor de todas las celdas es 1 y el bloque solicitado pertenece al subconjunto *Set* o el valor de todas las celdas es 0 y el bloque solicitado pertenece al subconjunto *Reset*. En caso contrario, la respuesta es "no presenta reuso". Seguidamente, se escribe en las celdas indexadas el valor 1, si el elemento solicitado pertenece al subconjunto *Set*, o el valor 0, si el elemento solicitado pertenece al subconjunto *Reset*. Este filtro no necesita una funcionalidad de borrado, porque el entrelazado de las consultas acerca de los elementos del subconjunto *Set* y *Reset*, ya provoca este comportamiento.

En cuanto a recursos utilizados es muy similar al filtro Básico, exceptuando por el número de funciones hash requeridas. El tamaño de la tabla es igual al filtro Básico, pero el filtro SetReset necesita el doble de funciones hash para poder trabajar con los dos subconjuntos. Esto es porque cuando se consulta un elemento, se indexa la tabla utilizando las funciones hash propias del subconjunto al que pertenece.

Capítulo 4

Metodología

En este capítulo se presenta la metodología empleada en el proyecto. En las dos primeras secciones se exponen el entorno de simulación *ChampSim* y la herramienta de síntesis. En la tercera sección se muestran los *benchmarks* seleccionados para llevar a cabo la experimentación. Por último, en las últimas secciones se presentan los casos de prueba realizados para verificar la correcta implementación de las estructuras y las métricas empleadas respectivamente.

4.1. Simulador *ChampSim*

Una de las tareas más importantes en la investigación en la arquitectura de computadores, como en otras muchas disciplinas, es el uso de simuladores para demostrar la validez de nuevas hipótesis y cuantificar las mejoras de nuevos diseños en términos de coste y/o rendimiento. También es la tarea que más tiempo consume. El simulador utilizado en este proyecto se trata de *ChampSim* [6], suministrado en distintas competiciones internacionales del campo de la arquitectura de computadores. Entre las más recientes se encuentra la 3ª competición de prebúsqueda de datos, DCP-3 [19], 1ª edición de prebúsqueda de instrucciones, IPC-1 [20], 2ª competición de reemplazo en cache, CRC-2 [21], y la 5ª edición de predicción de saltos, CBP-5 [22]. *ChampSim* emula un sistema cuyas características (número de cores, frecuencia, predictor de saltos, etc.) se encuentran definidas en un archivo JSON. Este archivo de configuración es modificable, por lo que fácilmente se pueden ajustar las características del sistema para establecer las deseadas. Por otro lado, el comportamiento de este sistema está programado mediante el lenguaje *C++*.

Para definir las características del sistema que se desea simular, se ha partido de la configuración inicial, proporcionada por el simulador en su versión actual, y a esta se le han modificado algunos aspectos relacionados con la jerarquía de memoria. Por lo tanto, el entorno simulado en este proyecto cuenta con una jerarquía de caches de tres

niveles, donde el algoritmo de reemplazo para los niveles de caches 1 y 2 es LRU y para el último nivel es SRRIP, cuyo funcionamiento se encuentra explicado en el Anexo A. El tamaño de bloque es de 64 bytes y el tamaño, conjuntos y vías de cada cache y del TLB, se encuentra detallado en la tabla 4.1.

Los distintos filtros Bloom estudiados en este proyecto; Básico extendido, A^2, SBF, SetReset, y las estructuras ReD y *Brain*, han sido implementadas en el mismo lenguaje de programación que el simulador, *C++*, e incorporadas en este para poder evaluar el rendimiento del sistema cuando cuenta con estas estructuras. Se encuentran situadas entre memoria principal y la cache de último nivel, con el fin de filtrar los bloques insertados en LLC, provenientes de memoria principal. A tal efecto, ha sido necesario realizar un estudio previo para conocer a fondo el funcionamiento del simulador, ya que apenas existe documentación, y la modificación de múltiples archivos del código fuente para poder incorporar dichas estructuras y automatizar el lanzamiento de las simulaciones.

Para compilar y ejecutar el simulador, se ha utilizado el compilador *gcc* versión 11.3 y el nivel 3 de optimización, junto con la herramienta *Make* para facilitar el proceso. El entorno donde se ha ejecutado el sistema simulado, cuenta con dos procesadores Kunpeng 920-4826 con 48 núcleos, cada uno, a 2.6 GHz y 320 GB de memoria RAM. Por otro lado, para poder llevar el control de versiones de los fuentes se ha utilizado la herramienta *GitHub*. La última versión del código implementado puede verse en el siguiente repositorio `https://github.com/AileonN/TFG_BloomFilter.git`.

	Caches			**TLB**	
	L1	**L2**	**LLC**	**L1**	**L2**
Tamaño	32 KB	256 KB	2 MB por core	4 KB	96 KB
Conjuntos	64	512	2048 por core	16	128
Vías	8	8	16	8	12

Tabla 4.1: Características de la jerarquía de memoria del sistema simulado

4.2. Carga de trabajo

Para llevar a cabo la parte experimental, *ChampSim* permite ejecutar *benchmarks* sobre el entorno simulado. De esta forma, se puede conocer y comparar el comportamiento y rendimiento del sistema, tras incorporar las distintas estructuras estudiadas en el proyecto, cuando se ejecutan sobre él aplicaciones de software actuales.

Existen múltiples conjuntos de *benchmarks* que recopilan aplicaciones enfocadas a diferentes segmentos de mercado, como tiempo real o *Cloud Computing*, creados por distintas organizaciones, grupos de investigación, etc. El propio comité organizador

de las competiciones internacionales selecciona los periodos de ejecución más representativos de cada aplicación usando la metodología *SimPoint* [23], extrae las trazas de dichos periodos y proporciona las trazas a los participantes.

En este proyecto se han estudiado los proporcionados por la organización SPEC para el estudio de prestaciones de estaciones de trabajo y servidores de cálculo. Más concretamente, en este proyecto se ha utilizado el subconjunto de las aplicaciones de SPEC CPU2006 [7] que usan la LLC de forma significativa (aquellas con un MPKI en LLC mayor que 1).

La Tabla 4.2 detalla las aplicaciones usadas y algunas de sus características obtenidas al ejecutar 500 millones de instrucciones sobre el entorno simulado.

Nombre de la traza	**Tasas LLC**		**MPKI**	**IPC**
	Accesos	**Fallos**		
429.mcf-217B	26416731	26349073	52,70	0,176
401.bzip2-277B	2608023	859900	1,72	1,202
403.gcc-16B	23447271	3353055	6,71	0,789
410.bwaves-1963B	9272086	9262238	18,52	0,356
433.milc-127B	10009116	10004433	20,01	0,523
434.zeusmp-10B	1765706	1023386	2,05	1,541
436.cactusADM-1804B	2584686	2570732	5,14	1,001
437.leslie3d-271B	3563171	3399951	6,80	0,994
450.soplex-247B	21676562	14953860	29,91	0,298
459.GemsFDTD-1320B	9857147	9316756	18,63	0,479
462.libquantum-714B	12999217	12999217	26,00	0,506
470.lbm-1274B	15086133	14750288	29,50	0,270
471.omnetpp-188B	10897724	8227053	16,45	0,364
473.astar-359B	19394540	18255967	36,51	0,161
481.wrf-816B	4150583	3904657	7,81	1,088
482.sphinx3-1100B	6365839	5504798	11,01	0,805
483.xalancbmk-127B	13991571	13125854	26,25	0,284

Tabla 4.2: Trazas seleccionadas y prestaciones obtenidas tras ser ejecutadas en el sistema mono-core base (LLC sin filtro)

4.3. Herramienta de síntesis

La FPGA donde se ha realizado la síntesis hardware es Xilinx Virtex 5 (xc5vlx20t-2ff323). Una FPGA [24] es un circuito integrado programable que contiene bloques lógicos, cuya interconexión y funcionalidad puede ser configurada en el momento mediante un lenguaje de descripción de hardware. En este caso, el lenguaje escogido es *VHDL*. Además, para verificar el correcto funcionamiento de los filtros antes de ser sintetizados, se ha utilizado la herramienta ModelSim [25]. El código de todos

los filtros en *VHDL* está disponible en el repositorio `https://github.com/AileonN/TFG_BloomFilter.git`.

4.4. Casos de prueba

Para verificar el correcto funcionamiento de la implementación de la estructura *Brain* y de las variantes de filtro Bloom estudiadas en este proyecto, se ha diseñado un conjunto de pruebas de caja negra. Concretamente, la técnica empleada ha sido la de tablas de decisión. El conjunto de pruebas se encuentra detallado en el Anexo C.

4.5. Métricas

En este trabajo se utilizan diferentes métricas para facilitar el entendimiento de los resultados obtenidos por el simulador, tras incorporar cada una de las estructuras estudiadas en este proyecto. Las métricas utilizadas son las siguientes:

- **Precisión en el recuerdo (PR)**: Mide la capacidad para recordar las últimas b direcciones de bloque solicitadas a memoria principal por LLC.

$$PR = \frac{Verdaderos\ Positivos + Verdaderos\ Negativos}{Verdaderos\ Positivos + Verdaderos\ Negativos + Falsos\ positivos + Falsos\ Negativos}$$

- **IPC**: Mide el número de instrucciones ejecutadas por ciclo.

$$IPC = \frac{Número\ de\ Instrucciones\ Simuladas}{Ciclos\ Totales}$$

- **MPKI en LLC**: Mide el número de fallos en LLC por cada mil instrucciones ejecutadas.

$$MPKI = \frac{Número\ de\ fallos\ en\ LLC \times 1000}{Número\ de\ Instrucciones\ Simuladas}$$

Capítulo 5

Resultados

En este capítulo se presentan y analizan los resultados obtenidos. En el caso del entorno mono-core simulado, se ejecutan 500 millones de instrucciones de cada una de las trazas seleccionadas, mientras que en el caso de la simulación multi-core, cada core ejecuta 50 millones. Estos valores han sido ajustado para que los resultados sean precisos y obtenidos en el menor tiempo de simulación posible.

Por otro lado, en todos los experimentos se ha definido el mismo escenario y es el siguiente. Se colocan los filtros de reuso que se desean analizar en el experimento entre LLC y memoria principal. De esta forma, los bloques analizados serán aquellos que provienen de memoria principal, solicitados debido a un fallo en LLC, para filtrar si se guardan o no en dicha cache. Entonces, antes de insertar uno de estos bloques se consulta el o los filtros utilizados en el experimento, para conocer si este presenta o no reuso. Si el filtro detecta que el bloque presenta reuso, el bloque se guarda en la cache con la mínima prioridad de expulsión. En caso contrario, queda almacenado con la prioridad más alta posible, para que sea seleccionado como el próximo bloque víctima. Además, para que ReD y los filtros Bloom presenten exactamente el mismo comportamiento, se incorpora que los filtros también introduzcan bloques que no presentan reuso a LLC con una probabilidad de 1/32. No obstante, no se tendrá en cuenta la respuesta de los filtros hasta que todas las vías del conjunto, donde se realiza la inserción del bloque, estén ocupadas. De esta forma, aprovechamos todo el espacio disponible de la cache.

Este capítulo se divide en 5 secciones. En la primera sección se detalla el análisis de la configuración de la estructura *Brain*, con la que se desea conocer el número óptimo de direcciones a recordar. Una vez obtenido este valor, en la siguiente sección se ajustan los parámetros de cada filtro en base a las métricas PR, IPC y MPKI. En la tercera sección, se comparan las prestaciones obtenidas por ReD, *Brain* configurado con la capacidad de recuerdo óptima y las mejores configuraciones de los filtros Bloom en un sistema mono-core. Además, se repite este mismo análisis para conocer las diferencias

entre las funciones hash H3 y MD5. En la cuarta sección, se compara el coste hardware de los filtros Bloom estudiados y finalmente, en la última sección, se realiza el mismo análisis que la sección cuatro, pero en un entorno multi-core.

5.1. Configuración de *Brain*: estudio de tamaño óptimo

La estructura *Brain* cuenta con un parámetro que denota el marco temporal de recuerdo, es decir, el número de direcciones de bloque que la estructura es capaz de recordar. Lo que se busca conseguir con este análisis es conocer el número óptimo de bloques solicitados a memoria principal de los que se quiere conocer si presentan o no reuso, para que el sistema presente las mejores prestaciones posibles. Por este motivo, se utiliza *Brain* que es una estructura ideal capaz de recordar exactamente el número de bloques establecido.

La figura 5.1 muestra los resultados obtenidos al simular, sobre el entorno *ChampSim*, la estructura *Brain* con diferentes capacidades de recuerdo. En este caso, la métrica empleada es MPKI, cuyo valor es una media del obtenido en cada una de las trazas evaluadas.

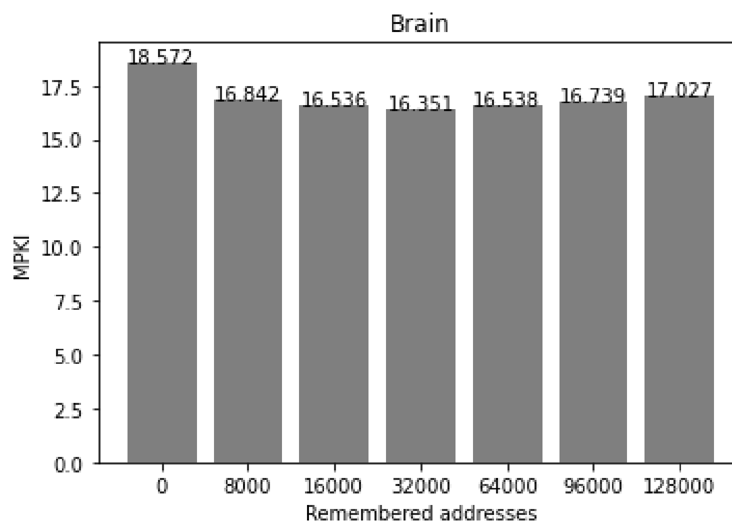

Figura 5.1: Estudio del tamaño óptimo de recuerdo para LLC

En base a los resultados obtenidos, el número ideal de direcciones recordadas es de 32000, puesto que es la configuración con menor MPKI. Este valor es equivalente a la máxima capacidad de recuerdo de LLC, porque con su tamaño de 2 MB puede almacenar hasta 32768 bloques de 64 B. Por otro lado, se puede observar que la degradación de prestaciones escala mucho más rápido cuando se aumenta la capacidad de recuerdo que cuando se disminuye. A pesar de ello, todas las configuraciones

analizadas mejoran respecto a la configuración base. En esta, *Brain* no recuerda ninguna dirección y por tanto, el sistema ignora esta estructura comportándose como si no estuviese presente.

5.2. Ajuste de filtros Bloom

Una vez conocido el número óptimo de direcciones de bloque, de las que se quiere saber si presentan o no reuso, se ha realizado un análisis para conocer la precisión de recuerdo de cada uno de los filtros, además de las prestaciones que presenta el sistema tras incorporar dichas estructuras. Más concretamente, se estudian distintas configuraciones de los filtros, obtenidas mediante la variación sus parámetros, para encontrar aquellas con las que se obtiene un mayor PR e IPC y por tanto, menor MPKI. Para obtener la precisión de recuerdo se compara la respuesta o clasificación obtenida por los filtros, con la de la estructura *Brain*. Adicionalmente, se repite este experimento en el Anexo D, para conocer la mejor alternativa con la que clasificar los bloques en los subconjuntos *Set* y *Reset* del filtro SetReset.

Los parámetros que permiten configurar los filtros, excepto el tamaño total que ocupa la estructura, varían dependiendo del tipo de filtro escogido. El tamaño total de la estructura viene determinado por su número de celdas y el tamaño de estas. Por ejemplo, si la estructura ocupa 32 KB, el filtro Básico extendido y SetReset, cuentan con una tabla de 2^{18} celdas de 1 bit, mientras que cada tabla del filtro A2 contará con 2^{17} celdas de 1 bit cada una. En el caso del filtro SBF, depende del número de bits que ocupe cada celda. Si el tamaño de estas es de 2 bits, su tabla estará formada por 2^{17} celdas de 2 bits. Los tamaños analizados para todas las estructuras estudiadas son; 32, 16, 8 y 4 KB. Limitamos 32 KB como tamaño máximo porque la estructura con la que comparamos el rendimiento de los filtros, ReD, ocupa 28 KB. Además, aunque se pudiese obtener un mayor rendimiento con un tamaño de filtro superior, el coste en área sería demasiado elevado.

De la misma forma, en el filtro Bloom extendido y en A2 se puede variar el número de hashes y el nivel máximo de llenado del filtro. En ambos, el número de hashes analizadas son 1, 3, 5 y 7, pero el nivel de llenado máximo del filtro Básico extendido varía entre 50 %, 40 % y 35 %, y el de A2 entre 40 %, 35 % y 30 %. Por otro lado, *SetReset* solo cuenta con un parámetro adicional al tamaño, que es el número de hashes utilizadas para indexar. Este varía entre 1, 2, 3, 4, 5 y 7. Por último, el filtro SBF permite configurar el tamaño de la celda, el número de hashes y el número de celdas borradas aleatoriamente por cada acceso al filtro. En cuanto al tamaño por celda, se han estudiado los valores 1, 2 y 3 bits. Mientras que el número de hashes escogidas

varía entre 1, 3 y 5, y el número de celdas borradas entre 1, 5 y 10. Se limita el número máximo de celdas borradas a 10 porque se necesita un generador de números aleatorios por cada celda, lo cual aumenta el coste energético y de área de la estructura.

5.2.1. Básico extendido

Análisis de precisión

En la figuras 5.2 se presentan los resultados de precisión obtenidos para el filtro Básico extendido. Los distintos patrones (círculo, triángulo y cruz) representan los diferentes niveles máximos de llenado, mientras que los colores representan el número de funciones hash utilizadas.

Lo que podemos observar con los resultados obtenidos es que utilizar una sola hash proporciona una precisión muy baja. Esto es debido a que se produce una gran cantidad de falsos positivos, ya que para determinar que un bloque presenta reuso, solamente debe aparecer el valor 1 en una celda. En cuanto al resto de configuraciones, estas presentan una precisión de recuerdo similar que aumenta con el tamaño de la estructura. Para facilitar la comparación, la figura 5.3 muestra un zoom de la parte alta de la figura 5.2.

La precisión aumenta entorno a un 1,6 % cada vez que se duplica el tamaño del filtro. Para cada tamaño, la mejor precisión se obtiene con borrados al 50 % y 5 funciones hash.

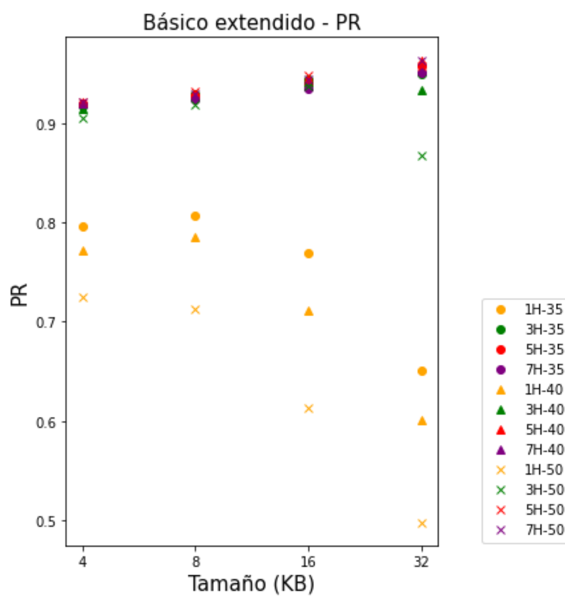

Figura 5.2: Precisión en el recuerdo: Filtro Básico Extendido

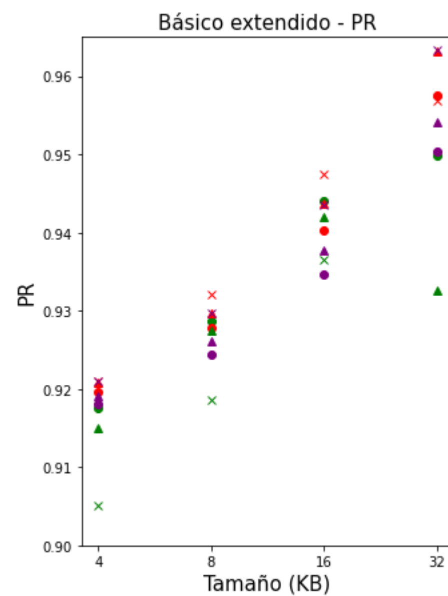

Figura 5.3: Precisión en el recuerdo: Filtro Básico Extendido, escala aumentada

Análisis de rendimiento

Las figuras 5.4 y 5.5 muestran los datos de IPC y MPKI para las configuraciones con mayor precisión. En primer lugar cabe destacar que las diferencias son pequeñas. La diferencia entre la mejor configuración y la peor es solo de un 1,2 % en MPKI y de un 0,45 % en IPC. El mejor resultado se obtiene con un tamaño de 32 KB, 5 funciones hash y borrado al 35 % de su capacidad.

Otra observación interesante es que no existe una correspondencia exacta entre tener una precisión de recuerdo alta y tener mejores valores de MPKI e IPC. Es decir, filtros con menor precisión consiguen mejores prestaciones en cuanto a MPKI e IPC. Esto nos lleva a analizar qué ocurre en el sistema cuando no insertamos bloques sin reuso a LLC con una probabilidad de 1/32, tal y como hacen los filtros implementados replicando las propuestas EAF y ReD. Esto es porque añadir bloques de forma aleatoria equivale a añadir falsos positivos al filtro, es decir, a disminuir su precisión. Para simplificar el análisis, este experimento solo se va a presentar para el tamaño de 16 KB.

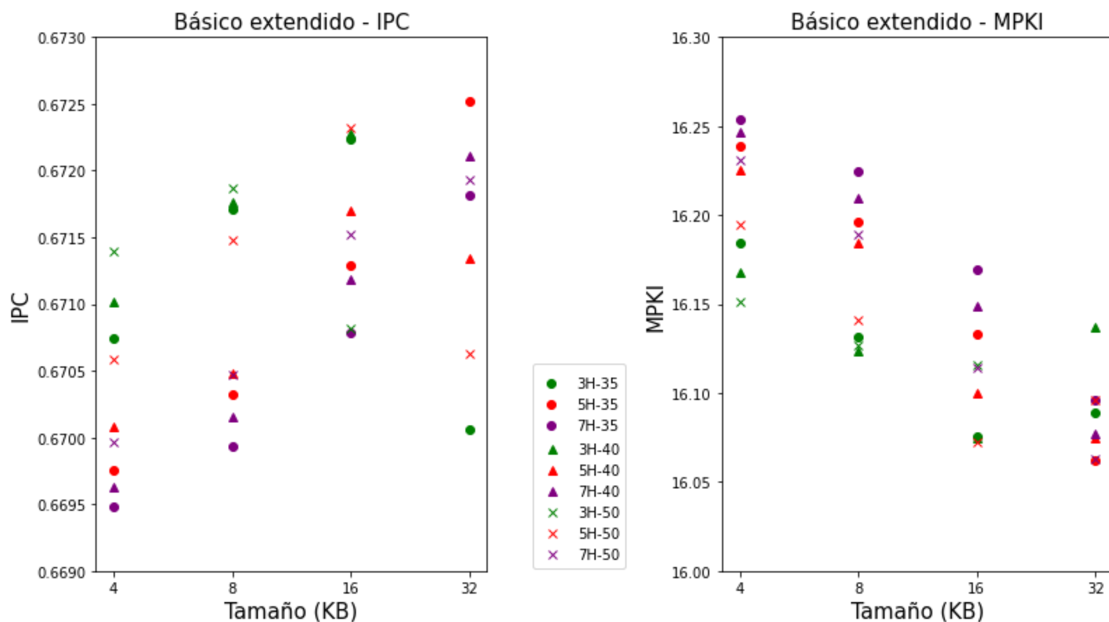

Figura 5.4: IPC: Filtro Básico Extendido, escala ajustada a los mejores resultados

Figura 5.5: MPKI: Filtro Básico Extendido, escala ajustada a los mejores resultados

En la figura 5.6 se comparan las dos alternativas que se desean estudiar: introducir en LLC basándonos solamente en la respuesta del filtro, e insertar, además, bloques que no presentan reuso con una probabilidad de 1/32. Esta segunda alternativa introduce los bloques con la prioridad de expulsión más baja, tal y como si presentase reuso.

El IPC obtenido por los sistemas que no introducen bloques de forma aleatoria es mejor al disminuir el número de funciones hash y al aumentar el nivel máximo de llenado del filtro. Ambos casos tienen en común que aumentan el número de falsos

positivos. Al realizar una consulta con menos funciones hash aumenta la probabilidad de encontrar 1´s por casualidad en todas las celdas consultadas y por tanto, aumentan los falsos positivos. Por otra parte, cuanto más llenamos el filtro también es mayor la probabilidad de encontrar todo 1´s por casualidad en todas las celdas consultadas.

Al añadir uno de cada 32 bloques sin reuso de forma aleatoria prácticamente se igualan las prestaciones de todas las configuraciones. Todo ello nos lleva a la misma conclusión, la cual es que es necesario introducir una pequeña cantidad de bloques no reusados a LLC, es decir, que el filtro presente falsos positivos para que el sistema tenga buenas prestaciones.

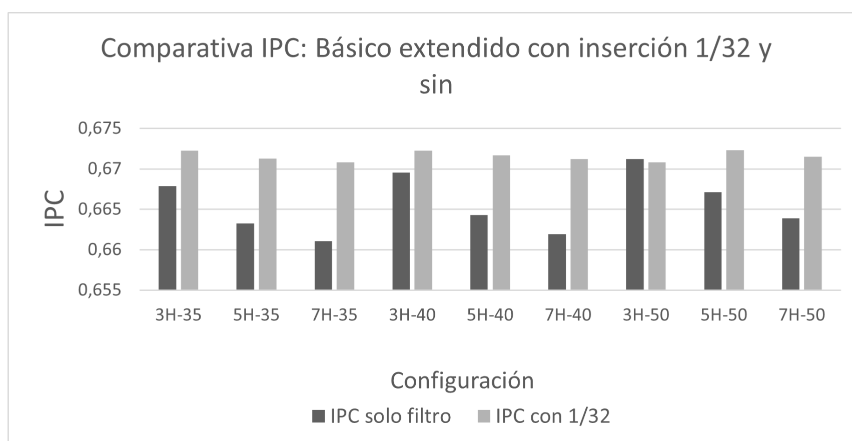

Figura 5.6: Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (Básico extendido)

5.2.2. A2

Análisis de precisión

En la figuras 5.7 y 5.8, se observan los resultados obtenidos para el filtro A2. Los parámetros de configuración, nivel máximo de llenado y el número de hashes, se representan en el gráfico de la misma forma que en la subsección anterior.

Al igual que ocurre en el filtro Básico extendido, utilizar una sola hash proporciona muy malos resultados en cuanto a PR. Los datos con 3 funciones hash son también claramente peores que en el filtro Básico. Esto es porque a pesar de analizar los mismos tamaños de estructura, este se reparte entre las dos tablas por las que está conformado. Por tanto, el número de celdas de cada tabla es menor y es más probable que las tres celdas consultadas coincidan con el valor 1. El resto de configuraciones presentan una alta precisión de recuerdo, la cual aumenta al mismo tiempo que el tamaño de la estructura, porque se tiene más capacidad para recordar más direcciones y llegar al número óptimo.

La precisión aumenta en torno a un 1.36 % cada vez que se duplica el tamaño del filtro. Además, para todos los tamaños la mejor configuración es utilizar 7 hashes y borrar cuando el nivel de llenado está al 40 %.

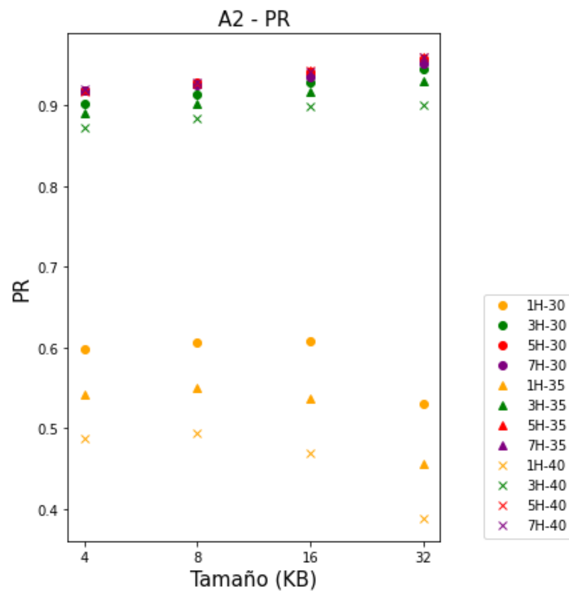

Figura 5.7: Precisión en el recuerdo: Filtro A2

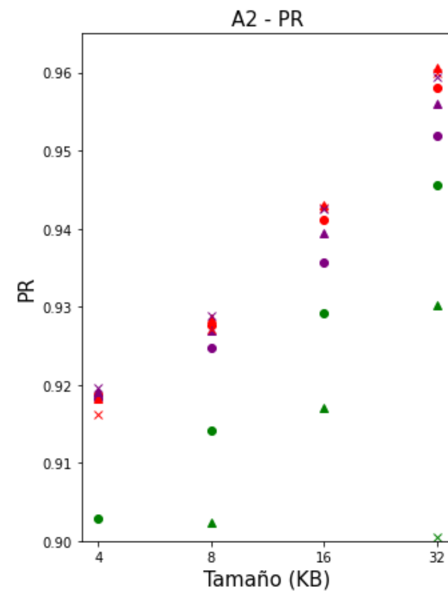

Figura 5.8: Precisión en el recuerdo: Filtro A2, escala aumentada

Análisis de rendimiento

A continuación obtenemos las tasas de IPC y MPKI, en el entorno simulado, para las configuraciones con mejor precisión. Estas se muestran en las figuras 5.9 y 5.10. Las diferencias entre todas las configuraciones son muy pequeñas. Más concretamente, la diferencia entre la mejor configuración y la peor es de un 1,3 % en MPKI y de un 0,49 % en IPC. El mejor resultado es obtenido utilizando 16 KB por tabla (32 KB en total), 5 hashes y borrando al 40 % de su capacidad. Al igual que en la subsección anterior, se puede ver claramente como las mejores tasas de IPC tampoco se corresponden con las configuraciones que tienen una mayor precisión de recuerdo. Por lo tanto, repetimos el mismo análisis que para el filtro Básico extendido, llegando a las mismas conclusiones (datos en la figura 5.11).

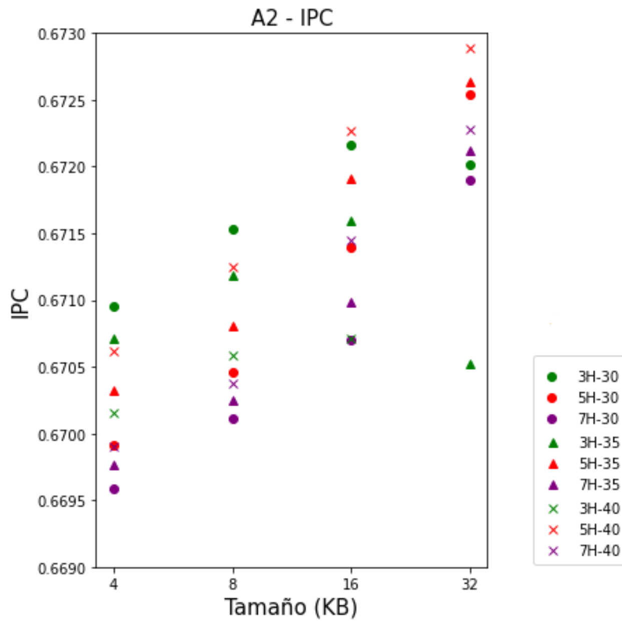

Figura 5.9: IPC: Filtro A2,
escala ajustada a los mejores resultados

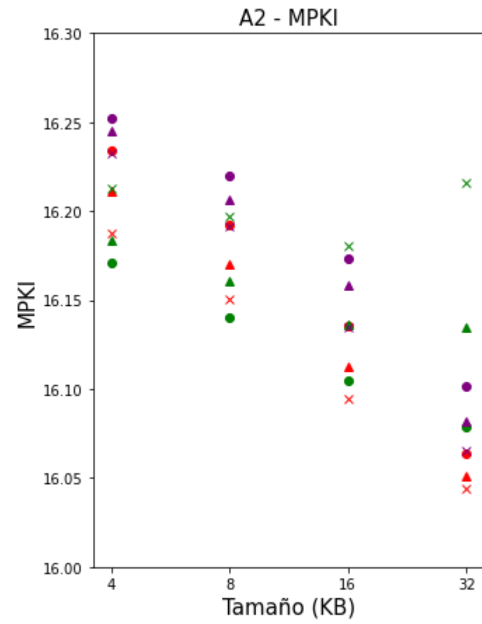

Figura 5.10: MPKI: Filtro A2,
escala ajustada a los mejores resultados

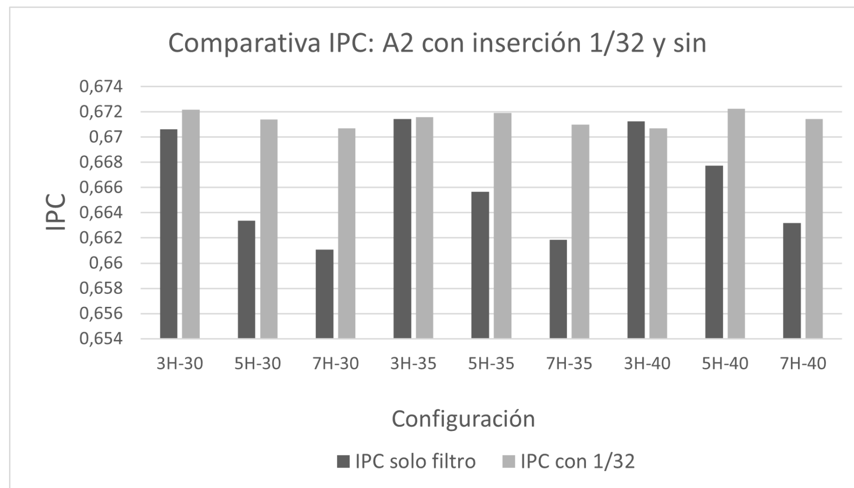

Figura 5.11: Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (A2)

5.2.3. SBF

Análisis de precisión

En la figura 5.12 se muestran los resultados obtenidos para el filtro SBF. Se añade la figura 5.13, que es un zoom de la parte alta de 5.12, con el fin de facilitar el análisis. En este caso, en el eje X no solo se representa el tamaño total de la estructura, sino también el número de bits por celda. Nótese que al aumentar el número de bits por celda se disminuye el número de entradas de la tabla para mantener el tamaño constante. Se utiliza el color amarillo, verde y rojo, para representar el número de funciones hash utilizadas y los símbolos círculo, cruz y triángulo para el número de celdas que son borradas aleatoriamente con cada acceso al filtro.

La primera conclusión que podemos extraer es que la precisión de recuerdo del filtro disminuye notablemente cuando las celdas están formadas por más de 1 bit, independientemente del tamaño total de la estructura. En general, la precisión aumenta al incrementar el número de funciones hash y el número de celdas borradas en cada inserción. Esto último está relacionado con la capacidad de recuerdo del filtro, ya que si solamente borramos una celda, el filtro es capaz de recordar más direcciones que el número óptimo. En consecuencia, se obtienen peores resultados.

Resumiendo, las mejores configuraciones se obtienen cuando el tamaño de la celda es de 1 bit, el número de celdas borradas es 10 y el número de hashes es 3 o 5. La precisión aumenta menos que en otros filtros cada vez que se duplica el tamaño del filtro, en torno a un 0,8 %.

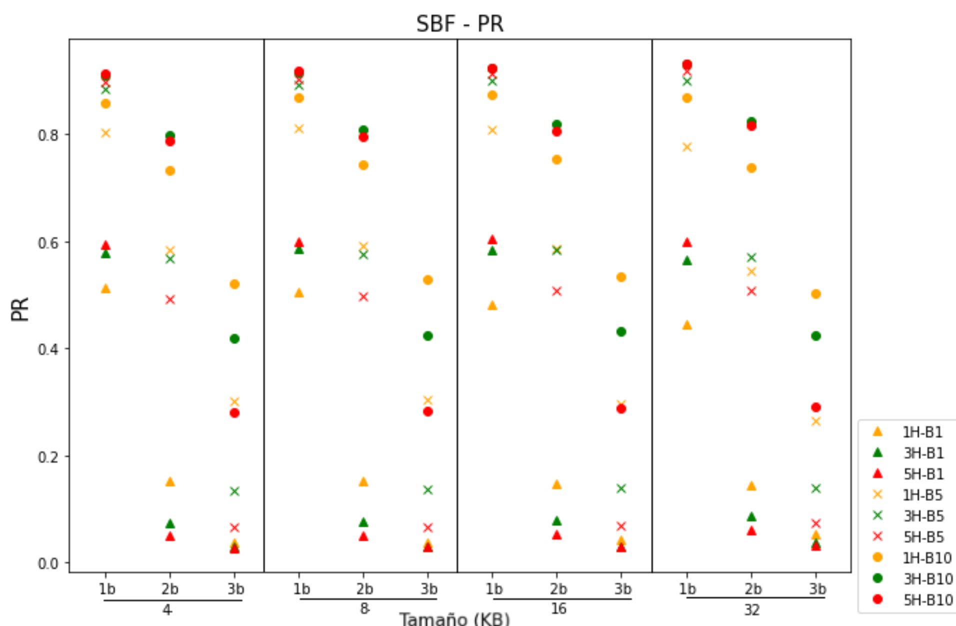

Figura 5.12: Precisión en el recuerdo: Filtro SBF

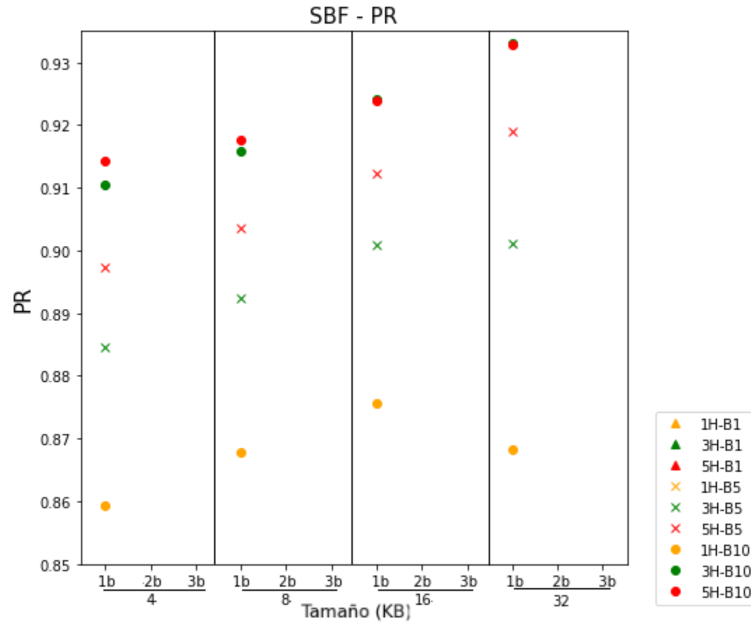

Figura 5.13: Precisión en el recuerdo: Filtro SBF, escala aumentada

Análisis de rendimiento

Se presentan las mejores tasas de IPC y MPKI obtenidas en las figuras 5.14 y 5.15 respectivamente. Se puede ver claramente como utilizar mas de un bit por celda no consigue estar entre las mejores tasas, ya que no consigue que ninguna configuración cuente con una precisión alta. Por otro lado, al igual que ocurre con el resto de filtros, las diferencias entre las mejores configuraciones de cada tamaño son pequeñas. La diferencia entre la mejor configuración y la peor es de un 1,08 % en MPKI y de un 0,47 % en IPC. La configuración con la que se obtienen mejores prestaciones es con un tamaño de 32 KB, celdas de 1 bit, 3 funciones hash y 10 celdas borradas por acceso.

Por otro lado, al igual que en el resto de filtros, las mejores prestaciones no son obtenidas con las configuraciones que presentan las tasas de PR mas altas. Por lo tanto, realizamos el mismo análisis que en las secciones anteriores (figura 5.16), donde se puede ver que incorporar bloques no reusados de forma aleatoria no aporta la mejora que se podía observar en el resto de filtros. Esto es debido a que el propio filtro ya genera un exceso de falsos positivos.

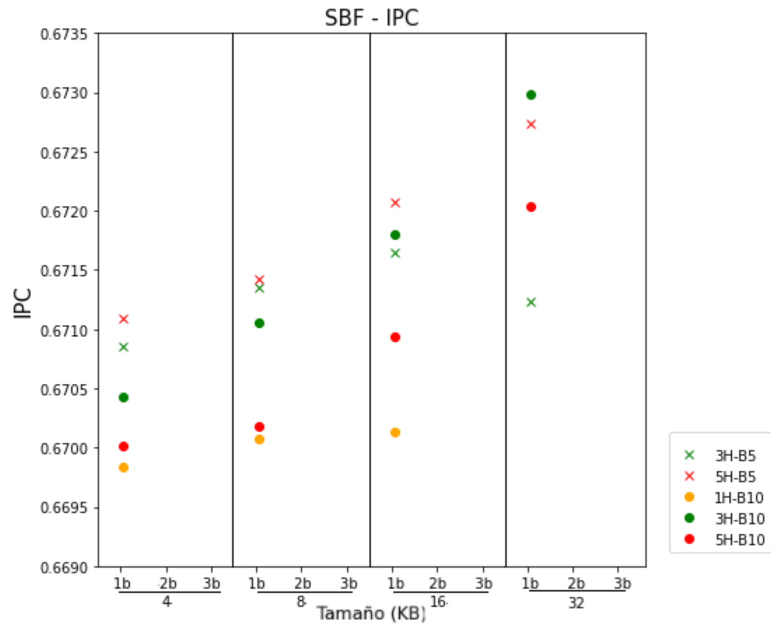

Figura 5.14: IPC: Filtro SBF, escala ajustada a los mejores resultados

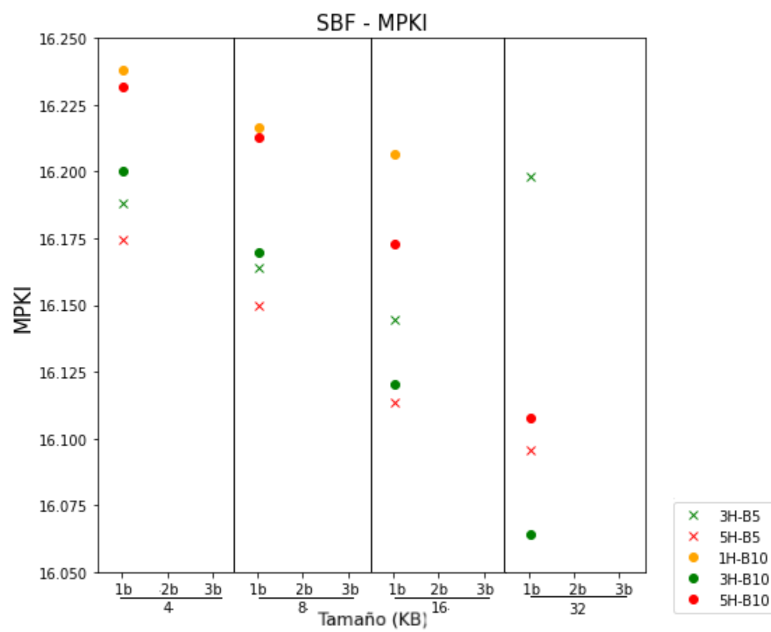

Figura 5.15: MPKI: Filtro SBF, escala ajustada a los mejores resultados

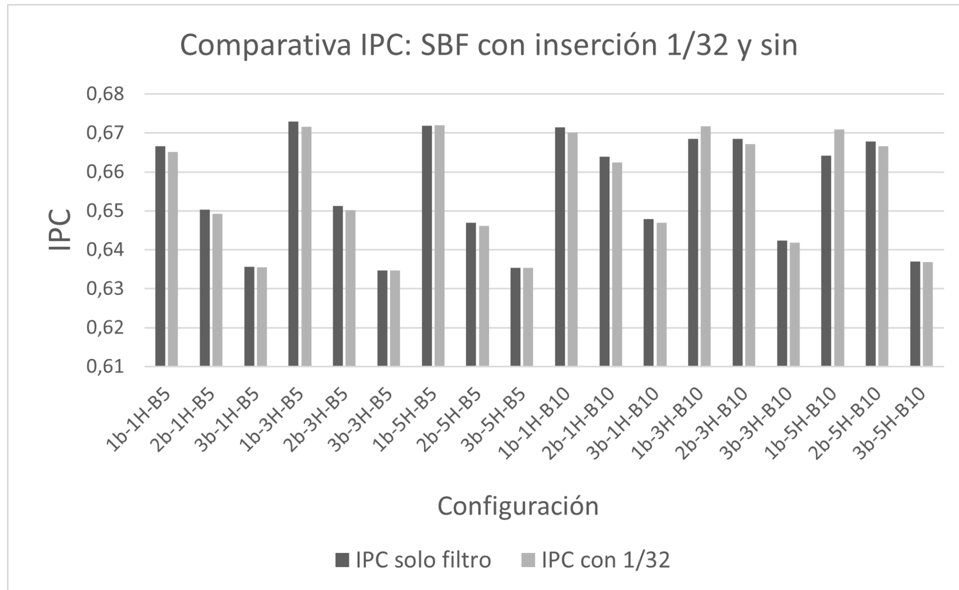

Figura 5.16: Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (SBF)

5.2.4. SetReset

Análisis de precisión

En la figura 5.17 se presentan los resultados obtenidos para el filtro SetReset. En el grafo se representa el número de funciones hash utilizadas para indexar cada subconjunto, *Set* y *Reset*, mediante un color. Por ejemplo, el color verde indica que se utilizan 2 funciones hash para cada subconjunto, por lo que la estructura cuenta con un total de 4 funciones.

En este caso, los valores de PR más altos son obtenidos a medida que aumenta el número de funciones hash utilizadas. El número de funciones por subconjunto se ha limitado a 7 para no aumentar en exceso su coste de implementación. Asimismo, se puede observar que a medida que aumenta el tamaño de la estructura, se necesita un mayor número de hashes para conseguir un buen resultado. Esto es, porque tener pocas hashes en un espacio cada vez mayor, permite almacenar más direcciones de las deseadas.

La precisión aumenta entorno a un 0,685 % cada vez que se duplica el tamaño del filtro. En cada tamaño, la mejor precisión se obtiene utilizando 7 hashes.

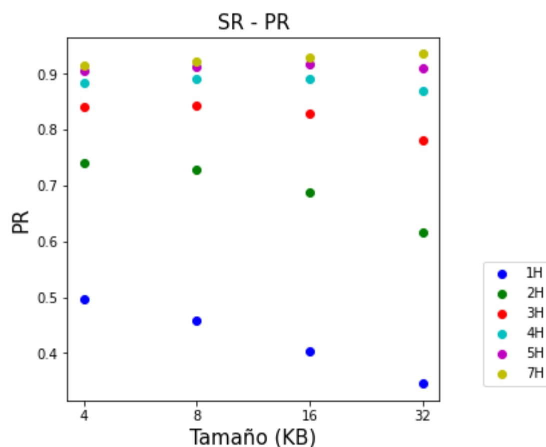

Figura 5.17: Precisión en el recuerdo: Filtro SetReset

Análisis de rendimiento

Por otro lado, en las figuras 5.18 y 5.19 se muestran las prestaciones obtenidas al incorporar este filtro sobre el sistema simulado. La diferencia entre el mejor resultado y el peor es pequeña, ya que esta es de un 0,65 % de IPC y un 1.28 % en MPKI. El mejor resultado se obtiene cuando la estructura tiene un tamaño de 32 KB y 7 hashes para indexar. En este caso, a medida que aumentamos el tamaño de la estructura si existe una mayor correlación entre presentar una tasa de precisión y prestaciones altas. Pero esto no se cumple para todos los tamaños, ya que es destacable que a medida que el tamaño de la estructura disminuye, es preferible utilizar un número de hashes inferior. Por lo tanto, se repite el mismo análisis que en las secciones anteriores (figura 5.20) llegando a las mismas conclusiones.

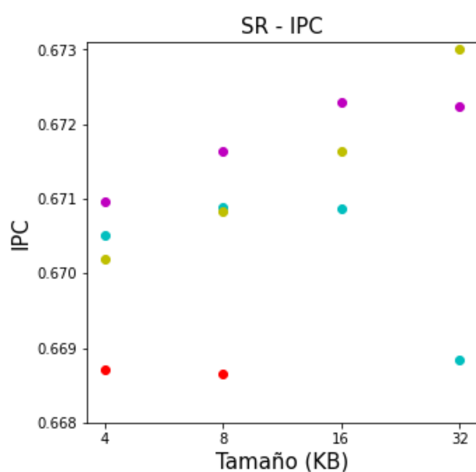

Figura 5.18: IPC: Filtro SetReset, escala ajustada a los mejores resultados

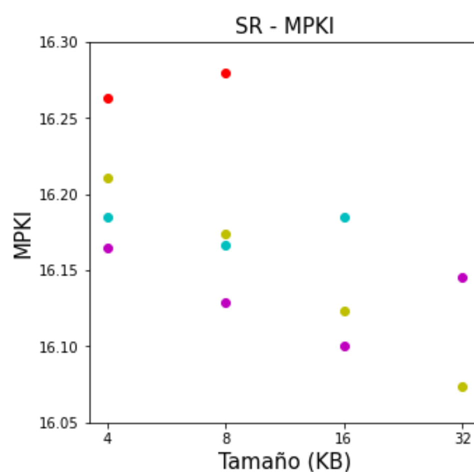

Figura 5.19: MPKI: Filtro SetReset, escala ajustada a los mejores resultados

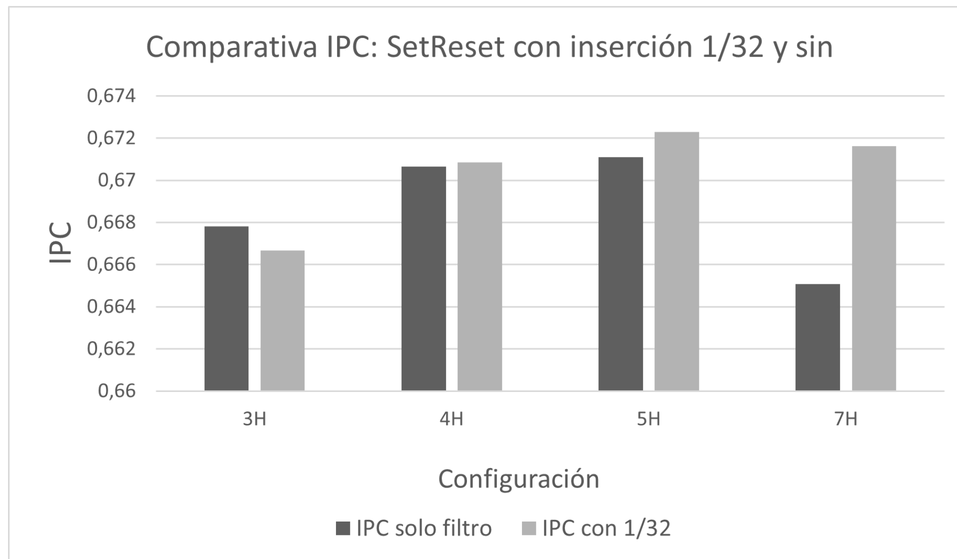

Figura 5.20: Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (SetReset)

5.3. Comparación global de prestaciones

En los siguientes experimentos se desea conocer y comparar las prestaciones obtenidas por todas las estructuras estudiadas en un sistema mono-core. Para simplificar el análisis, se limitan las configuraciones de filtros Bloom seleccionadas, escogiendo la configuración, por cada tamaño y filtro, con la que se obtiene una tasa de IPC más alta.

Los resultados de este análisis se presentan en las figuras 5.21 y 5.23, donde el tipo de filtro Bloom se representa mediante un color y las configuraciones de parámetros con un símbolo. Por otro lado, los resultados obtenidos con *Brain* y ReD, se representan mediante una línea de color negra y azul respectivamente.

El primer dato a destacar es que todas las configuraciones seleccionadas superan a Brain. Esto es debido a que Brain no introduce falsos positivos. En cuanto a los filtros Bloom, si nos fijamos en los resultados obtenidos para un tamaño concreto, podemos ver que la diferencia entre las distintas configuraciones es muy pequeña. Más concretamente, la diferencia entre el mejor resultado y el peor, para los tamaños 32, 8 y 4 KB, está entorno 0,067 % de IPC y 0,157 % de MPKI, mientras que para 16 KB la diferencia es de 0,036 % de IPC y 0,257 % de MPKI. Lo mismo ocurre si comparamos los resultados entre los distintos tamaños. A pesar de que las prestaciones aumentan al duplicar el tamaño, su diferencia es pequeña. El mejor resultado para 4 KB disminuye solamente en un 0,239 % de IPC y 0,625 % de MPKI frente al mejor resultado de 32 KB. Lo cual significa que, disminuyendo considerablemente el coste hardware de la estructura, podemos conseguir igualmente tasas de prestaciones altas.

Por otro lado, el filtro de reuso ReD consigue ganar al resto de estructuras, incluso si el tamaño de ambas es el mismo. A pesar de ello, la diferencia de la mejor configuración de cada tamaño con ReD no es muy notable. Con 32 KB tenemos una diferencia de 0,19 % de IPC y 0,23 % de MPKI, que aumenta en torno al 0,08 % cada vez que se divide por 2 el tamaño de la estructura.

Desde el punto de vista global, utilizando ReD se consigue aumentar las prestaciones hasta un 5,74 % en IPC respecto a un sistema sin filtro, mientras que utilizando la mejor configuración de los filtros Bloom se alcanza una mejora del 5,54 % de IPC.

5.3.1. Función hash alternativa

Hasta ahora, la hash utilizada en simulación para indexar las tablas de los filtros Bloom es MD5. Esta función es muy compleja y su implementación hardware presenta un coste en área muy elevado, lo cual es un problema a la hora de utilizarla en un entorno real. Por este motivo, se ha implementado una hash de bajo coste, H3 [10], con la que se ha realizado el mismo análisis. De esta forma, se busca conocer si existe alguna diferencia, en cuanto a prestaciones, entre utilizar una u otra hash. Este análisis puede verse en las figuras 5.22 y 5.24.

Comparando las figuras 5.21 y 5.22, se concluye que es indiferente utilizar la hash MD5 o H3, puesto que el sistema presenta un rendimiento similar. Por lo tanto, la implementación hardware de los filtros Básico extendido, A2, SBF y SetReset, incorpora H3 como función hash.

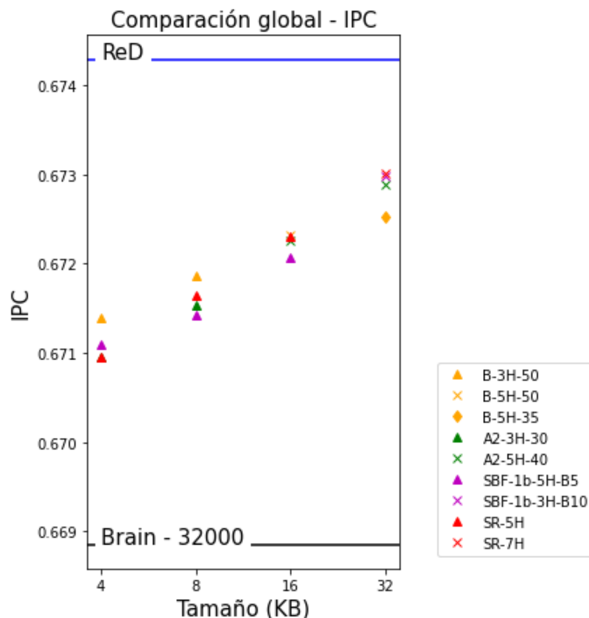

Figura 5.21: Comparación del IPC obtenido con todas las estructuras estudiadas

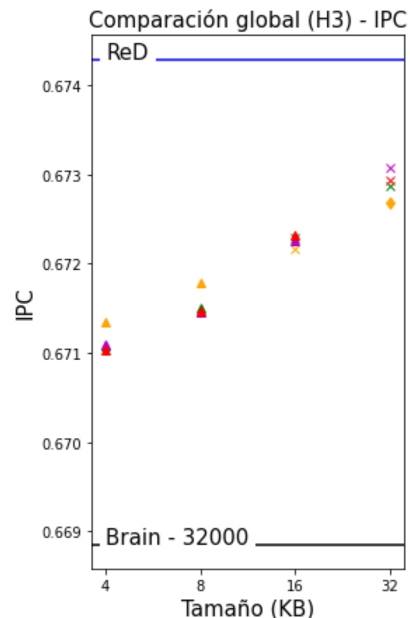

Figura 5.22: Comparación del IPC obtenido con todas las estructuras estudiadas utilizando H3

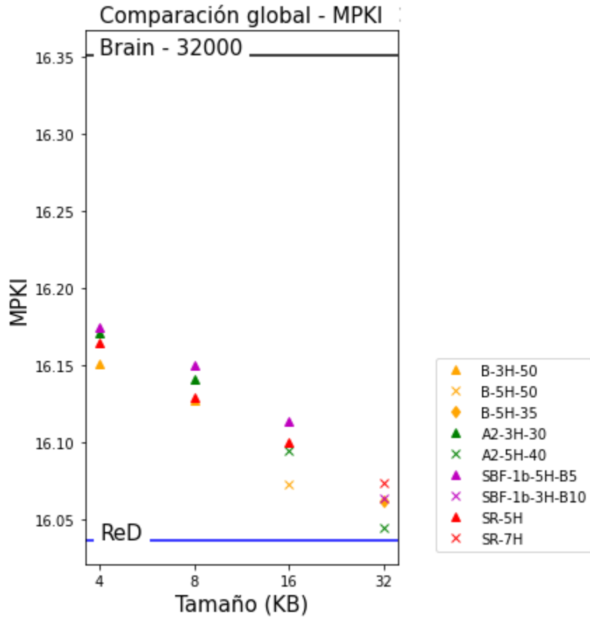

Figura 5.23: Comparación del MPKI obtenido con todas las estructuras estudiadas

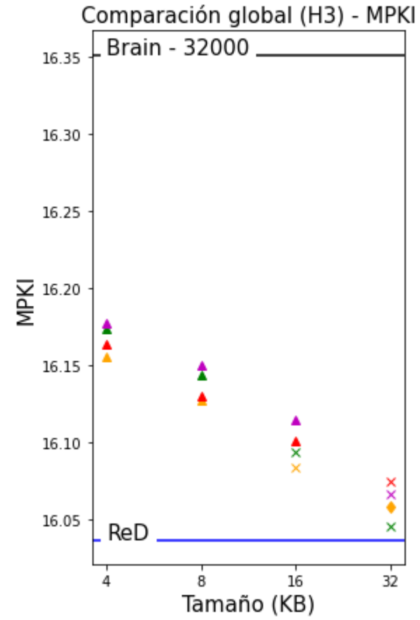

Figura 5.24: Comparación del MPKI obtenido con todas las estructuras estudiadas utilizando H3

5.4. Coste en área

En esta sección se analiza el impacto en área de cada uno de los filtros Bloom. Para analizar el coste en área de los distintos filtros se ha usado una herramienta de síntesis para FPGAs.

Para implementar cada función hash H3 se necesitan tantas puertas AND como bits tiene la dirección de bloque (ver tabla 5.1) y una unidad menor de puertas XOR. Además, el número de bits de entrada de cada puerta es equivalente al tamaño del índice con el que se indexa la tabla de los filtros. Por otro lado, para implementar la tabla de los filtros es necesario un registro flip-flop por celda de un bit y un multiplexor por cada puerto de acceso, cuyo número de entradas es equivalente al número de celdas de la tabla. Se necesita un puerto de acceso por cada lectura que realiza una función hash. En cuanto al hardware requerido para implementar la funcionalidad de borrado, este varía dependiendo del tipo de filtro. El filtro Básico extendido y A2 necesitan llevar la cuenta del número de celdas cuyo valor es 1. Para ello, es necesario un registro donde se almacena este valor y un comparador y sumador por cada función hash.

Por último, el filtro SBF necesita un generador de números aleatorios por cada celda que se quiere borrar. Aunque en este caso, se ha escogido LFSR [26] que los genera de forma pseudo-aleatoria. La implementación realizada consta de 3 puertas XNOR y tantos registros flip-flop como bits tiene el índice. También se necesita un restador y un

comparador por cada celda borrada. Estos comparadores se utilizan para comprobar que el valor de la celda no desborde al restar. Además, para determinar si un bloque presenta o no reuso (todas las celdas tienen un valor mayor a 0), se necesita también un comparador por celda accedida.

A pesar de que el código diseñado permite generar cualquier configuración de los filtro Bloom implementados (variando tamaño, número de hashes, etc.), la antigüedad de la FPGA no permite sintetizar estructuras de gran tamaño. Es por ello, por lo que los filtros finalmente sintetizados cuentan con un número de entradas y hashes reducido. De esta forma, se busca comparar el coste hardware de cada uno de ellos de forma cualitativa, analizando una configuración representativa aunque no real. Esta configuración, junto con los resultados, puede verse en la tabla 5.1.

	Tipo de filtro			
	Básico extendido	A2	SetReset	SBF
Tamaño de la dirección de bloque (bits)	32	32	32	32
Tamaño estructura (bits)	256	256	256	256
Tamaño celda (bits)	1	1	1	2
Número de celdas por tabla (Filas)	256	128	256	128
Número de funciones hash	2	2	4	2
Condición de borrado	Nº celdas con valor 1 >= 128	Nº celdas con valor 1 >= 128	———	4 celdas por acceso
Registros Flip-Flop	264	264	256	288
Puertas XOR (8-bit)	62	62	124	62
Puertas XOR (1-bit)	0	0	0	14
Puertas AND (8-bit)	64	64	128	64
Multiplexor 256:1	2	0	4	0
Multiplexor 128:1	0	4	0	4
Comparadores	2	2	0	136
Restadores (2-bit)	0	0	0	131
Sumadores (8 bits)	2	2	0	0
%LUTs (Tablas de consulta)	6 %	7 %	12 %	26 %
Tiempo de ciclo del camino crítico (ns)	17.090	18.561	18.633	19.032

Tabla 5.1: Características de los filtros sintetizados y resultados obtenidos

Observando los resultados obtenidos, se puede ver como el análisis teórico anteriormente realizado, se corresponde con el de los resultados de la síntesis. El filtro que menor coste en área presenta, 6 % de LUTs utilizadas, es el filtro Básico extendido. Esto es, claramente, porque es el que menos recursos hardware necesita. A este le siguen A2 y SetReset con un 7 % y un 12 % respectivamente. En el caso de A2, el aumento se debe a que incluye dos multiplexores adicionales para poder consultar dos celdas en cada una de sus tablas. El aumento de SetReset se debe a la duplicación de funciones hash. Por último, con una gran diferencia respecto al resto de filtros, 26 % de LUTs utilizadas, se encuentra SBF. Este aumento se debe a la funcionalidad de borrado. A pesar de que solamente es necesario un restador y un comparador de dos bits por cada función hash, el sintetizador añade estos componentes por cada una de las celdas que componen su tabla. También presenta un consumo adicional debido a los 4 LFSR.

Partiendo de estos resultados, se puede estimar la escalabilidad de cada filtro. A medida que se duplica el tamaño de cada estructura, el número de registros flip-flop aumenta de la misma forma. Aunque en el caso del filtro SBF, si el incremento de tamaño viene dado por un mayor número de celdas y no por un aumento en el tamaño de la celda, también se incrementa el número de comparadores y restadores de la misma forma. Sin embargo, el área necesaria para implementar las funciones hash no depende del tamaño de los filtros. Por lo tanto, el área ocupada por las funciones hash toma menor relevancia contra mayor es el tamaño de la estructura.

5.5. Multi-core

Adicionalmente, se desea conocer y comparar las prestaciones obtenidas cuando se incorpora la estructura ReD y cada una de las mejores configuraciones de filtros Bloom en un sistema multi-core. En este caso, el sistema simulado cuenta con 4 cores donde cada uno de ellos cuenta con su propio filtro. Para este experimento, se desea que cada uno de los cores ejecute una traza distinta. Por lo tanto, se han agrupado las 17 trazas seleccionadas del paquete SPEC CPU2006 en grupos de 4 de forma aleatoria, para formar las 10 combinaciones con las que se han llevado a cabo los experimentos.

En la figura 5.25 y 5.26 se presentan las tasas de IPC y MPKI conseguidas por cada uno de los filtros de reuso y por ReD. Cabe destacar tres diferencias significativas respecto a los resultados en sistemas mono-core: 1) La diferencia entre todos los resultados obtenidos es significativa. Dentro de un mismo tamaño, la mayor diferencia de IPC entre el mejor resultado y el peor se consigue con 32 KB y varía un 2,7 %. En cuanto a MPKI, la mayor variación, 2,11 %, se obtiene con 16 KB. Comparando los resultados entre los distintos tamaños, el mejor resultado de 4 KB y el de 32 KB presentan una diferencia de un 3,18 % en IPC y 0,45 % en MPKI. 2) Los valores de IPC y MPKI no correlan de forma tan directa como en el sistema mono-core. Esto es debido a que las reducciones en MPKI se traducen en aumentos de IPC de forma distinta para cada aplicación de una mezcla. 3) Existe una configuración del filtro A2 y otra del Básico extendido que logran superar las prestaciones de ReD. Más concretamente, el mejor resultado presenta una diferencia del 1,65 % en IPC y 0,47 % en MPKI respecto a ReD. A pesar de ello, solo se consigue alcanzar cuando el tamaño del filtro (32 KB) es equivalente al de ReD (28 KB).

Como resultado final, utilizando el filtro A2 de 32 KB, 5 funciones hash y 40 % como nivel máximo de llenado, conseguimos mejorar un 12,21 % de IPC y un 15 % de MPKI frente al sistema multi-core base (sin ningún filtro).

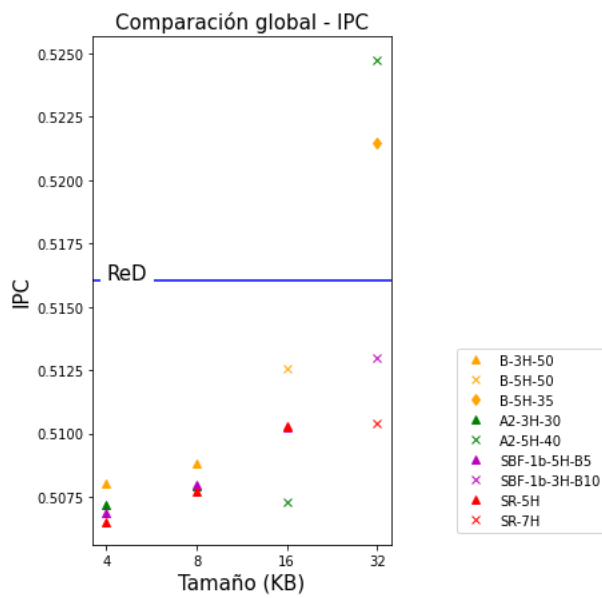

Figura 5.25: Resultados IPC multi-core de las mejores configuraciones de filtros y ReD

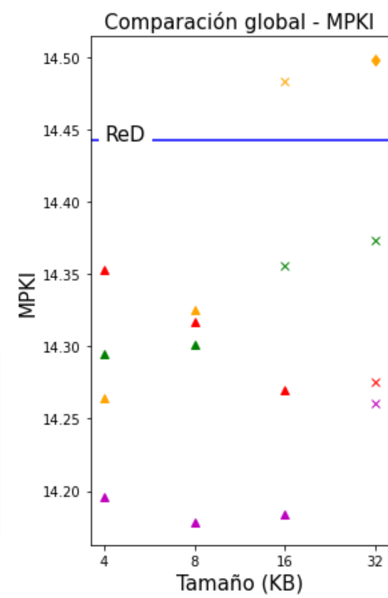

Figura 5.26: Resultados MPKI multi-core de las mejores configuraciones de filtros y ReD

Capítulo 6

Conclusiones

El acceso a memoria es uno de los principales cuellos de botella en un sistema. Es por ello, por lo que establecer un uso eficiente del espacio disponible en cada nivel de la jerarquía de memoria permitirá mejorar notablemente las prestaciones del sistema.

En este trabajo, se ha estudiado el uso de filtros de reuso para paliar esta problemática. Estos se encargan de filtrar el contenido proveniente de memoria principal hacia LLC, para que solamente aquellos bloques con una alta probabilidad de reuso sean almacenados. Para ello, el filtro debe recordar direcciones de bloque que han provocado fallo en LLC recientemente. Se han analizado varias implementaciones basadas en filtros Bloom y se han comparado con ReD, otra implementación que utiliza una estructura de datos organizada de forma similar a una cache. En concreto, los filtros Bloom analizados son el Básico extendido, A2, SBF y una nueva propuesta, SetReset. También se ha estudiado el filtro de reuso ideal *Brain*, con el que se busca calificar la precisión de recuerdo de cada uno de los filtros Bloom estudiados. Como en las propuestas previas que usan filtros Bloom y ReD, se ha añadido la inserción aleatoria en LLC de algunos bloques que no presentan reuso.

Los filtros de reuso estudiados se han implementado sobre el entorno de simulación *Champsim* en el que se han llevado a cabo diferentes experimentos. En el primer análisis, se ha utilizado *Brain* para conocer el número óptimo de direcciones de bloque que los filtros deben recordar para que el sistema presente las mejores prestaciones. Dado que la menor tasa de MPKI se consigue recordando 32000 direcciones, se ha establecido este valor como número óptimo de recuerdo. Seguidamente, se han ajustado los parámetros de cada filtro en base a las métricas PR, IPC y MPKI, donde claramente se ha observado que no existe una correspondencia exacta entre tener una precisión de recuerdo alta y tener mejores valores de MPKI e IPC. Es decir, filtros con menor precisión consiguen mejores prestaciones en cuanto a MPKI e IPC. Esto nos ha llevado a analizar qué ocurre en el sistema cuando no insertamos bloques sin reuso a LLC de forma aleatoria. Los resultados obtenidos indican que es necesario introducir una

pequeña cantidad de bloques no reusados a LLC, es decir, que el filtro presente falsos positivos, para que el sistema tenga buenas prestaciones.

Posteriormente, se han analizado las tasas de IPC y MPKI obtenidas por las mejores configuraciones de cada filtro Bloom en sistemas mono-core y multi-core, y se han comparado con las de ReD y *Brain*. Aunque los mejores resultados en mono-core son obtenidos utilizando ReD, la diferencia respecto a los filtros Bloom es pequeña. Más concretamente, con el mejor resultado de 32 KB tenemos una diferencia de 0,19 % de IPC y 0,23 % de MPKI, que aumenta en torno al 0,08 % a medida que disminuye a la mitad el tamaño de la estructura. Con todo ello, ReD consigue aumentar las prestaciones hasta un 5,74 % de IPC, mientras que utilizando la mejor configuración de los filtros Bloom se alcanza un aumento del 5,54 % en IPC. En cuanto al sistema multi-core, existen dos configuraciones de filtros Bloom que consiguen mejorar las prestaciones de ReD. La mejor de ellas es utilizando el filtro A2 de 32 KB, 5 funciones hash y 40 % como nivel máximo de llenado. Con esta configuración conseguimos mejorar un 12,21 % de IPC y un 15 % de MPKI las prestaciones del sistema multi-core base (sin ningún filtro).

Por otro lado, se han sintetizado los filtros Bloom estudiados sobre una FPGA utilizando una función hash de bajo coste, en este caso H3, con el fin de conocer su coste hardware. Con los resultados obtenidos en la síntesis se observa que el coste hardware del filtro Básico extendido y el de A2 son los más bajos de todos (6 % y 7 % de LUTs utilizadas respectivamente). A estos les sigue SetReset (12 %) cuyo aumento se debe a la duplicación de funciones hash y por último, con gran diferencia, SBF (26 %) cuyo coste se debe a la gran cantidad de recursos utilizados para la implementación de su funcionalidad de borrado.

Queda como trabajo futuro repetir el análisis de coste hardware usando una herramienta de síntesis VLSI como por ejemplo, Synopsys [27], en vez de sintetizar sobre una FPGA, tal y como se ha realizado en el proyecto. Este análisis nos daría una idea más fiel del coste de las alternativas en un entorno similar al de un procesador. La especificación realizada en *VHDL* es válida para este tipo de herramientas.

Bibliografía

[1] D Munix. Intel® Xeon® Processor Scalable Family Technical Overview, 2017.

[2] Javier Díaz, Teresa Monreal, Pablo Ibáñez, José M Llabería, and Víctor Viñals. Red: A reuse detector for content selection in exclusive shared last-level caches. *Journal of Parallel and Distributed Computing*, 125:106–120, 2019.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[4] Myungkeun Yoon. Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge and Data Engineering*, 22:134–138, 2010.

[5] Fan Deng and Davood Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 25–36, New York, NY, USA, 2006. Association for Computing Machinery.

[6] ChampSim. GitHub - ChampSim/ChampSim: ChampSim repository.

[7] Agustín Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Víctor Viñals-Yúfera. Memory hierarchy characterization of spec cpu2006 and spec cpu2017 on the intel xeon skylake-sp. *PLOS ONE*, 14(8):1–24, 08 2019.

[8] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 355–366, New York, NY, USA, 2012. Association for Computing Machinery.

[9] R Rivest. The MD5 Message-Digest Algorithm. RFC 1321, RFC Editor, April 1992.

[10] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997.

[11] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A tagless coherence directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 423–434, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Lailong Luo, Deke Guo, Richard Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, PP, 04 2018.

[13] Evangelos Pournaras, Martijn Warnier, and Frances MT Brazier. A generic and adaptive aggregation service for large-scale decentralized networks. *Complex Adaptive Systems Modeling*, PP, 11 2013.

[14] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: An efficient data structure for static support lookup tables. volume 15, pages 30–39, 01 2004.

[15] Paolo Palmieri, Luca Calderoni, and Dario Maio. Spatial bloom filters: Enabling privacy in location-aware applications. In Dongdai Lin, Moti Yung, and Jianying Zhou, editors, *Information Security and Cryptology*, pages 16–36, Cham, 2015. Springer International Publishing.

[16] Cen Zhiwang, Xu Jungang, and Sun Jian. A multi-layer bloom filter for duplicated url detection. In *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, volume 1, pages V1–586–V1–591, 2010.

[17] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: An efficient data structure for static support lookup tables. volume 15, pages 30–39, 01 2004.

[18] F. Chang and Wu-chang Feng. Approximate caches for packet classification. volume 4, pages 2196 – 2207 vol.4, 04 2004.

[19] Third Data Prefetching Championship (Workshop with ISCA-2019) – TCCA, 02 2019.

[20] IPC - The 1st Instruction Prefetcher , 05 2020.

[21] THE 2ND CACHE REPLACEMENT CHAMPIONSHIP – Co-located with ISCA June 2017, 06 2017.

[22] Championship Branch Prediction, 2016.

[23] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, page 318–319, New York, NY, USA, 2003. Association for Computing Machinery.

[24] Wikipedia contributors. Field-programmable gate array.

[25] Wikipedia contributors. Modelsim.

[26] Russell. Linear Feedback Shift Register for FPGA, 06 2022.

[27] RTL Design and Synthesis.

[28] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, jun 2003.

[29] Fpga architecture white paper. 2006.

Lista de Figuras

1.1. Diagrama de Gantt. Tiempo dedicado al proyecto por tareas. 3

2.1. Representación gráfica del funcionamiento del filtro de reuso en la jerarquía de memoria 7

5.1. Estudio del tamaño óptimo de recuerdo para LLC 21
5.2. Precisión en el recuerdo: Filtro Básico Extendido 23
5.3. Precisión en el recuerdo: Filtro Básico Extendido, escala aumentada . . 23
5.4. IPC: Filtro Básico Extendido, escala ajustada a los mejores resultados . 24
5.5. MPKI: Filtro Básico Extendido, escala ajustada a los mejores resultados 24
5.6. Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (Básico extendido) 25
5.7. Precisión en el recuerdo: Filtro A2 26
5.8. Precisión en el recuerdo: Filtro A2, escala aumentada 26
5.9. IPC: Filtro A2, escala ajustada a los mejores resultados 27
5.10. MPKI: Filtro A2, escala ajustada a los mejores resultados 27
5.11. Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (A2) . 27
5.12. Precisión en el recuerdo: Filtro SBF 28
5.13. Precisión en el recuerdo: Filtro SBF, escala aumentada 29
5.14. IPC: Filtro SBF, escala ajustada a los mejores resultados 30
5.15. MPKI: Filtro SBF, escala ajustada a los mejores resultados 30
5.16. Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (SBF) 31
5.17. Precisión en el recuerdo: Filtro SetReset 32
5.18. IPC: Filtro SetReset, escala ajustada a los mejores resultados 32
5.19. MPKI: Filtro SetReset, escala ajustada a los mejores resultados 32
5.20. Comparación de IPC cuando incorporamos al sistema la introducción de bloques reusados con probabilidad 1/32 o cuando no lo hacemos (SetReset) 33

5.21. Comparación del IPC obtenido con todas las estructuras estudiadas . . 34
5.22. Comparación del IPC obtenido con todas las estructuras estudiadas utilizando H3 34
5.23. Comparación del MPKI obtenido con todas las estructuras estudiadas . 35
5.24. Comparación del MPKI obtenido con todas las estructuras estudiadas utilizando H3 35
5.25. Resultados IPC multi-core de las mejores configuraciones de filtros y ReD 39
5.26. Resultados MPKI multi-core de las mejores configuraciones de filtros y ReD 39

D.1. Precisión en el recuerdo obtenida al clasificar con la dirección de bloque 60
D.2. Precisión en el recuerdo obtenida al clasificar con la función hash . . . 60

Lista de Tablas

4.1. Características de la jerarquía de memoria del sistema simulado 17
4.2. Trazas seleccionadas y prestaciones obtenidas tras ser ejecutadas en el sistema mono-core base (LLC sin filtro) 18

5.1. Características de los filtros sintetizados y resultados obtenidos 36

C.1. Tabla de decisión: Filtro Bloom extendido 54
C.2. Casos de prueba: Filtro Bloom extendido 55
C.3. Tabla de decisión: Filtro A2 55
C.4. Casos de prueba: Filtro A2 55
C.5. Tabla de decisión: Filtro SBF 56
C.6. Casos de prueba: Filtro SBF 56
C.7. Tabla de decisión: Filtro SetReset 56
C.8. Casos de prueba: Filtro SetReset 57
C.9. Características de los filtros con los que se han realizado los casos de prueba 57
C.10. Tabla de decisión: *Brain* 57
C.11. Casos de prueba: *Brain* 58

Anexos

Anexo A

Algoritmos de reemplazo: LRU y SRRIP

En este anexo se detalla el funcionamiento del algoritmo de reemplazo utilizado para los niveles de cache 1 y 2, LRU, y para el último nivel, SRRIP.

LRU

El algoritmo de reemplazo *Least Recently Used*, LRU, expulsa de la cache el elemento que ha pasado mas tiempo sin ser utilizado. Para ello, etiqueta cada vía de la cache, mediante $\log_2(\textit{Número de vías})$ bits, cuyo valor inicial es igual al número de vías. Por cada acceso a la cache, los bits de la vía accedida toman el valor mínimo, 0, mientras que los del resto de vías aumentan su valor en 1. De esta forma, el elemento expulsado será aquel que esté almacenado en la vía cuyos bits tengan mayor valor, y por tanto, el menos utilizado.

SRRIP

El algoritmo de reemplazo *Static Re-Reference Interval Prediction*, SRRIP, prioriza para expulsión aquellos bloques que acaban de ser insertados a la cache, prediciendo que no volverán a ser utilizados. Para ello, se etiqueta cada vía de la cache con 2 bits, cuyo valor inicial es el máximo posible, es decir, 3. Cuando se produce un acierto, los bits de la vía en la que se encuentra almacenado toman el valor 0. Por el contrario, si se produce un fallo, primeramente se escoge el bloque víctima, y después se inserta el nuevo bloque con prioridad 2. Para escoger el bloque víctima, se busca entre todas las vías del conjunto, aquella cuyos bits tienen el máximo valor, 3. En caso de no encontrarla, se aumenta en uno el valor de todas las vías, y se repite la búsqueda hasta encontrar la víctima.

Anexo B

Funciones hash

En el presente anexo se detalla el funcionamiento de las hashes estudiadas, MD5 y H3.

B.1. MD5

MD5 es una de las funciones hash más utilizadas con la que, independientemente del tamaño de la entrada, se obtiene un valor de 128 bits. A continuación, se detallan los 5 pasos del algoritmo, donde el término "palabra"hace referencia a un valor de 4 bytes y cuya entrada inicial se denota como $m0\ m1\ ...\ m\{b-1\}$, siendo b la longitud de esta. De manera aclaratoria, en el presente trabajo la entrada a la función hash se corresponde con la dirección de los bloques estudiados y la salida, el índice con el que se indexan las tablas de los filtros.

Paso 1: Adición de bits de extensión

La entrada, o mensaje, será extendida hasta que su longitud en bits sea congruente con 448, módulo 512. Esto es, si se le resta 448 a la longitud del mensaje tras este paso, se obtiene un múltiplo de 512. Esta extensión se realiza siempre, incluso si la longitud del mensaje es ya congruente con 448, módulo 512.

La extensión se realiza como sigue: un solo bit, con valor 1, se añade al mensaje, y después se añaden bits con el valor 0 hasta que la longitud en bits del mensaje extendido se haga congruente con 448, módulo 512. En todos los mensajes se añade al menos un bit y como máximo 512.

Paso 2: Adición de bits de longitud

Un entero de 64 bits que representa la longitud inicial b del mensaje se concatena por el final al resultado del paso anterior. En el supuesto, no deseado, de que b sea mayor que 2^{64}, entonces sólo los 64 bits de menor peso de b serán utilizados.

En este punto el mensaje resultante tiene una longitud que es un múltiplo exacto de 512 bits, en otras palabras, la longitud del mensaje es múltiplo de 16 palabras. Denotamos con M[0 ... N-1] las palabras del mensaje resultante, donde N es múltiplo de 16.

Paso 3: Inicialización del *buffer*

Un *buffer* de cuatro palabras (A, B, C, D) se utiliza para calcular la hash del mensaje, donde cada una de las letras A, B, C, D representa un registro de 32 bits. Estos registros se inicializan con los siguientes valores hexadecimales:

palabra A: 0x67452301
palabra B: 0xEFCDAB89
palabra C: 0x98BADCFE
palabra D: 0x10325476

Paso 4: Procesado del mensaje en bloques de 16 palabras

Primero, se definen cuatro funciones auxiliares que toman como entrada tres palabras de 32 bits y su salida es una palabra de 32 bits.

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

Los operadores $\oplus, \wedge, \vee, \neg$ son las funciones XOR, AND, OR y NOT respectivamente.

En cada posición de cada bit, F actúa como un condicional: si X, entonces Y; si no, Z. Las funciones G, H e I son similares a la función F, ya que actúan "bit a bit en paralelo"para producir sus salidas de los bits de X, Y y Z, dado que si cada bit correspondiente de X, Y y Z son independientes y no sesgados, entonces cada bit de G(X,Y,Z), H(X,Y,Z) e I(X,Y,Z) serán independientes y no sesgados.

Este paso utiliza una tabla de 64 elementos T[1 ... 64] construida con la función seno, siendo T[i] el elemento i-ésimo de esta tabla, que es igual a la parte entera del valor absoluto del seno de i 4294967296 veces, e i un valor en radianes.

Se realiza lo siguiente:

```
--------------------------------------------------------------
 /* Procesar cada bloque de 16 palabras. */
para i = 0 hasta N/16-1 hacer
   /* Copiar el bloque 'i' en X. */
   para j = 0 hasta 15 hacer
     hacer X[j] de M[i*16+j].
   fin para /* del bucle 'j' */
 /* Guardar A como AA, B como BB, C como CC, y D como DD. */
 /* Ronda 1. */
 /* [abcd k s i] denotarán la operación
      a = b + ((a + F(b, c, d) + X[k] + T[i]) <<< s). */
 /* Hacer las siguientes 16 operaciones. */
 [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
 [ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
 [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
 [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
 /* Ronda 2. */
 /* [abcd k s i] denotarán la operación
     a = b + ((a + G(b, c, d) + X[k] + T[i]) <<< s). */
 /* Hacer las siguientes 16 operaciones. */
 [ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
 [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
 [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
 [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
 /* Ronda 3. */
 /* [abcd k s t] denotarán la operación
     a = b + ((a + H(b, c, d) + X[k] + T[i]) <<< s). */
 /* Hacer las siguientes 16 operaciones. */
 [ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
 [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
 [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
 [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
 /* Ronda 4. */
 /* [abcd k s t] denotarán la operación
     a = b + ((a + I(b, c, d) + X[k] + T[i]) <<< s). */
```

```
/* Hacer las siguientes 16 operaciones. */
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
/* Ahora realizar las siguientes sumas. (Este es el incremento de cada
   uno de los cuatro registros por el valor que tenían antes de que
   este bloque fuera inicializado.) */
   A = A + AA
   B = B + BB
   C = C + CC
   D = D + DD
fin para /* del bucle en 'i' */
-----------------------------------------------------------------
```

Paso 5: Salida

El valor hash del mensaje es la salida producida por A, B, C y D. Comenzando por el byte de menor peso, A, y acabando con el byte de mayor peso, D.

B.2. H3

H3 es una función hash eficiente, en cuanto al hardware requerido para su implementación, ya que solamente utiliza operaciones XOR y AND. A continuación se detalla su funcionamiento.

Denotamos $A = \{0, 1, ..., 2^i - 1\}$ como el espacio de entrada, $B = \{0, 1, ..., 2^j - 1\}$ como el espacio de salida y Q el conjunto de todas las posibles ixj matrices booleanas, siendo i el tamaño en bits de la entrada y j el de la salida. Para un $q \in Q$ y $x \in A$, siendo $q(k)$ la cadena de bits de la k-ésima fila de la matriz q y $x(k)$ el k-ésimo bit de x, la función hash $h(x) : A \longrightarrow B$ se define como,

$$h(x) = x(1) \wedge q(1) \oplus x(2) \wedge q(2) \oplus ... \oplus x(i) \wedge q(i)$$

donde $\wedge$ denota la operación AND bit a bit y $\oplus$ la operación XOR.

De manera aclaratoria, en el presente trabajo la entrada de la función hash son las direcciones de bloque, mientras que la salida son los índices con los que se indexa la tabla de los filtros.

Anexo C

Pruebas de caja negra: Tablas de decisión

Una prueba de caja negra es un tipo de prueba de software directa. Existen múltiples técnicas, como particiones de equivalencia o tablas de decisión, que son las utilizadas en este proyecto. Una tabla de decisión se compone de condiciones, representadas con la letra C, y todas las posibles acciones generadas por el entrelazado de las condiciones, representadas con la letra A. A partir de esta tabla, se diseña el conjunto de casos de prueba.

Las tablas C.1, C.3, C.5, C.7 y C.10, se corresponden con las tablas de decisión de cada uno de los filtros estudiados y *Brain*, respectivamente. En caso de no estar marcada ninguna acción para una determinada condición, significa que dicho escenario no puede ser dado.

Las tablas C.2, C.4, C.6, C.8 y C.11, muestran los casos de prueba para unos filtros con las características de la tabla C.9 y para *Brain*.

C1: Bloque conocido	S	S	N	N
C2: Condición de borrado cumplida	S	N	N	S
A1: Detecta reuso		X		
A2: Inserta (recuerda)			X	X
A3: Provoca borrado				X

Tabla C.1: Tabla de decisión: Filtro Bloom extendido

ID	Entrada			Salida	
	Dirección de bloque	Nº de celdas ocupadas	Estado inicial del almacenamiento	Estado final del almacenamiento	Detección de reuso
1	0x80000000	0	No guarda dicha dirección	Inserta 2 1's en la tabla, lo recuerda	No reusado
2	0x80000000	2	Si guarda dicha dirección	Igual al inicial, no se modifica	Reusado
3	0x40000000	4	No guarda dicha dirección	Borrado, todo 0's	No reusado

Tabla C.2: Casos de prueba: Filtro Bloom extendido

C1: Bloque conocido por la tabla *active1*	S	S	S	S	N	N	N	N
C2: Bloque conocido por la tabla *active2*	S	S	N	N	S	S	N	N
C2: Condición de borrado cumplida	S	N	S	N	S	N	S	N
A1: Detecta reuso		X		X	X	X		
A2: Inserta (recuerda)					X	X	X	X
A3: Provoca borrado				X	X		X	

Tabla C.3: Tabla de decisión: Filtro A2

ID	Entrada			Salida	
	Dirección de bloque	Nº de celdas ocupadas en *active1*	Estado inicial de *active1/2*	Estado final de *active1/2*	Detección de reuso
1	0x80000000	2	Solo la guarda *active1*	Igual al inicial, no se modifica	Reusado
2	0x40000000	2	No guardan dicha dirección	Inserta 2 1's en la *active1*, lo recuerda e intercambia *active1* por *active2* borrando el contenido del nuevo *active1*	No reusado
3	0x80000000	2	Solo la guarda *active2*	Inserta 2 1's en la *active1*, lo recuerda e intercambia *active1* por *active2* borrando el contenido del nuevo *active1*	Reusado
4	0x80000000	2	La guardan ambas	Igual al inicial, no se modifica	Reusado

Tabla C.4: Casos de prueba: Filtro A2

C1: Bloque conocido	S	S	N	N
C2: Valor de la celda borrada mayor a 0	S	N	S	N
A1: Detecta reuso	X	X		
A2: Celda borrada disminuye su valor en 1	X		X	
A2: Insertar en la celda indexada el máximo valor posible	X	X	X	X

Tabla C.5: Tabla de decisión: Filtro SBF

ID	Entrada			Salida		
	Dirección de bloque	Estado inicial del almacenamiento	Valor celdas borradas	Estado final del almacenamiento	Valor final celdas borradas	Detección de reuso
1	0x80000000	No guarda dicha dirección	[0,0,0,0]	Inserta 2 3's en la tabla, lo recuerda.	[0,0,0,0]	No reusado
2	0x8000000	Si guarda dicha dirección	[1,1,1,1]	Igual al inicial, no se modifica	[0,0,0,0]	Reusado
3	0x40000000	No guarda dicha dirección	[1,0,0,0]	Inserta 2 3's en latabla, lo recuerda	[0,0,0,0]	No reusado
4	0x4000000	Si guarda dicha dirección	[0,0,0,0]	Igual al inicial, no se modifica	[0,0,0,0]	Reusado

Tabla C.6: Casos de prueba: Filtro SBF

C1: Bloque del subconjunto Set	S	S	N	N
C2: Bloque conocido	S	N	S	N
A1: Detecta reuso	X		X	
A2: Inserta "1"		X		
A3: Inserta "0"				X

Tabla C.7: Tabla de decisión: Filtro SetReset

ID	Entrada		Salida	
	Dirección de bloque	Estado inicial del almacenamiento	Estado final del almacenamiento	Detección de reuso
1	0x80000000 (Subconjunto Set)	No guarda dicha dirección	Inserta 2 1's en la tabla, lo recuerda	No reusado
2	0x80000001 (Subconjunto Reset)	No guarda dicha dirección	Inserta 2 0's en la tabla, lo recuerda	No reusado
3	0x80000000 (Subconjunto Set)	Si guarda dicha dirección	Igual al incial, no se modifica	Reusado
4	0x80000001 (Subconjunto Reset)	Si guarda dicha dirección	Igual al inicial, no se modifica	Reusado

Tabla C.8: Casos de prueba: Filtro SetReset

	Tipo de filtro			
	Básico extendido	A2	SetReset	SBF
Tamaño de la dirección de bloque (bits)	32	32	32	32
Tamaño estructura (bits)	256	512	256	512
Tamaño celda (bits)	1	1	1	2
Número de celdas por tabla	256	256	256	256
Número de funciones hash	2	2	4	2
Número de funciones hash	2	2	4	2
Condición de borrado	Nº celdas ocupadas >= 4	Nº celdas ocupadas >= 4	———	4 celdas por acceso

Tabla C.9: Características de los filtros con los que se han realizado los casos de prueba

C1: Bloque conocido	S	S	N	N
C2: Pertenece a los b bloques más recientes	S	N	N	S
A1: Detecta reuso	X			
A2: Inserta (recuerda)			X	
A3: Actualiza su marca temporal		X	X	

Tabla C.10: Tabla de decisión: *Brain*

ID	Entrada			Salida		
	Dirección de bloque	Estado inicial del almacenamiento	Pertenece a los 4 bloques más recientes	Estado final del almacenamiento	Pertenece a los 4 bloques más recientes	Detección de reuso
1	0x80000000	No conocida	No	Guarda la dirección y actualiza su marca temporal	Si	No reusado
2	0x8000000	Conocida	Si	Nada	Si	Reusado
3	0x80000000	Conocida	No	Actualiza su marca temporal	Si	No reusado

Tabla C.11: Casos de prueba: *Brain*

Anexo D

SetReset: Subconjuntos

El filtro SetReset divide los bloques analizados en dos subconjuntos, el subconjunto *Set* y el subconjunto *Reset*. En este trabajo, se han estudiado dos alternativas con las que se determina el subconjunto al que pertenece cada bloque, con el fin de utilizar aquella con la que se obtiene una clasificación más equitativa. Esto permite que el filtro recuerde con mayor precisión. La primera alternativa utiliza el bit de menor peso de las direcciones de bloque analizadas. De esta forma, si el valor de este bit es 1, el bloque pertenece al subconjunto *Set* y si es 0, al subconjunto *Reset*. La segunda alternativa aplica a la dirección de bloque una función hash y clasifica, de la misma forma que la primera alternativa, con el bit de menor peso del valor obtenido.

Para conocer si existe alguna diferencia entre ambas alternativas, se ha realizado el mismo experimento que el apartado 2.4 del capítulo 5, donde se evalúa la precisión de recuerdo, utilizando las dos alternativas estudiadas. En este análisis, se varía tanto el tamaño de la estructura, como el número de funciones hash utilizadas para indexar por cada subconjunto, las cuales son representadas mediante colores distintos. En las figuras D.1 y D.2 pueden verse los resultados obtenidos.

Observando los resultados, se puede ver claramente que no existe ninguna diferencia entre utilizar una u otra alternativa. Por tanto, se ha escogido la alternativa que trabaja solamente con la dirección de bloque, ya que utilizar una función hash aumenta el coste hardware requerido.

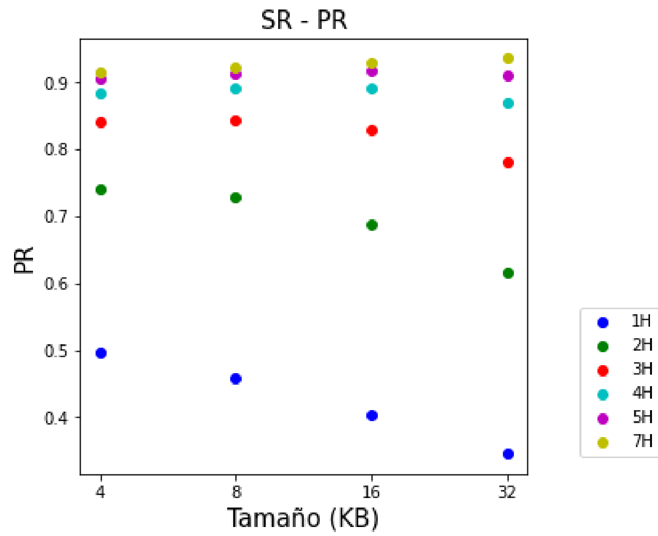

Figura D.1: Precisión en el recuerdo obtenida al clasificar con la dirección de bloque

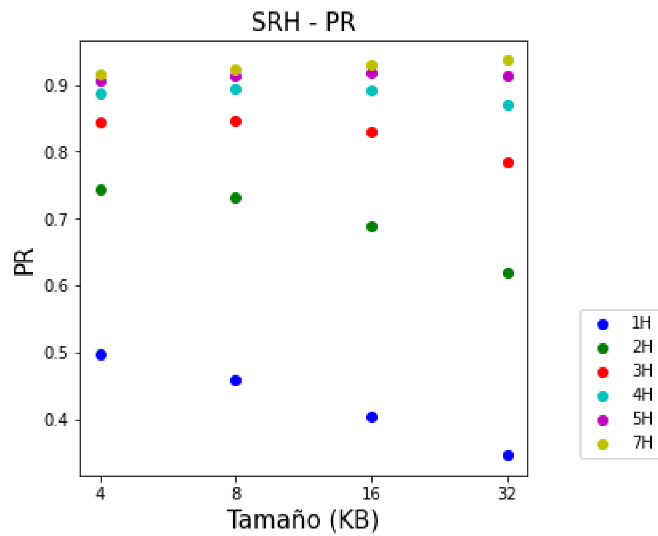

Figura D.2: Precisión en el recuerdo obtenida al clasificar con la función hash