Facultad de Ciencias

ANÁLISIS DE RENDIMIENTO DE UNA CACHE BICAMERAL EN ARQUITECTURAS VECTORIALES

(Performance Analysis of a Bicameral Cache in Vector Architectures)

Trabajo de Fin de Máster
para acceder al

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

Autora: Susana Rebolledo Ruiz

Directores: Borja Pérez Pavón
y Jose Luis Bosque Orero

Febrero – 2024

RESUMEN

Este trabajo presenta el diseño, modelado, implementación y evaluación de una propuesta de memoria cache para un procesador vectorial que segrega las referencias escalares y vectoriales en dos particiones con distintas características. Esta cache, denominada Cache Bicameral, está específicamente orientada a mejorar el rendimiento de las aplicaciones vectorizadas, no sólo suprimiendo las posibles interferencias causadas por las instrucciones escalares sobre los datos vectoriales, sino también disponiendo los elementos de los vectores de forma consecutiva para garantizar su continuidad. Adicionalmente, se incluye una opción de *prefetching* o rellenado de datos, que permite poblar las líneas de la cache vectorial con el fin de optimizar el rendimiento aprovechando la localidad espacial en las referencias vectoriales. Para la implementación se ha utilizado el simulador Cavatools, que soporta la extensión vectorial de RISC-V. El estudio y la evaluación de la propuesta respecto a una cache convencional se ha realizado mediante la simulación de cinco benchmarks distintos con diferentes longitudes de vector arquitectural. Los resultados indican que la cache propuesta mejora el rendimiento siempre y cuando tanto el peso de las referencias vectoriales en la aplicación como el tamaño de los datos que estas manejan sean lo suficientemente elevados, lo cual son dos hipótesis razonables en aplicaciones vectoriales. Por su parte, el *prefetching* ha resultado ser siempre una ventaja añadida.

Palabras clave: jerarquía de memoria, procesador vectorial, *prefetching*, extensión vectorial RISC-V, arquitectura de computadores.

ABSTRACT

This paper presents the design, modelling, implementation, and evaluation of a proposed cache memory for a vector processor that separates scalar and vector references into two partitions with different characteristics. This cache, called Bicameral Cache, is specifically aimed at improving the performance of vectorised applications, not only by suppressing the possible interferences caused by scalar instructions on vector data, but also by arranging vector elements consecutively to guarantee their continuity. Furthermore, a prefetching, or data prefilling, option is included to populate the vector cache lines to optimise performance by exploiting the spatial locality in the vector references. The Cavatools simulator which supports the RISC-V vector extension was used for the implementation. To evaluate the proposal against a conventional cache, simulations of five benchmarks with a variety of architectural vector lengths have been conducted. The results indicate that the proposed cache is advantageous when the weight of vector references in the application and the size of the data they handle are sufficiently large both of which are reasonable assumptions for vector applications. Additionally, prefetching consistently provides an added value.

Keywords: memory hierarchy, vector processor, prefetching, RISC-V vector extension, computer architecture.

AGRADECIMIENTOS

A Borja, por su inmensa ayuda y dedicación.

A Jose Luis, por guiarme y confiar siempre en mí.

To Peter, for his valuable insights.

Índice

Capítulo 1. Introducción

En este primer capítulo se introducen los motivos que impulsan el desarrollo de este trabajo, así como su finalidad y pasos necesarios para su realización. También se ofrece un breve resumen de la estructura del documento.

1.1. Motivación

Una de las principales soluciones para hacer frente a la creciente demanda en capacidad de cómputo que requieren las aplicaciones modernas es la vectorización. Esta técnica, que permite explotar el paralelismo a nivel de datos, no es exclusiva de los supercomputadores y sistemas de alto rendimiento, sino que está presente, también, en la mayoría de procesadores de propósito general del mercado [1].

Sin embargo, si bien las instrucciones vectoriales, al realizar una misma operación sobre varios datos de manera simultánea, pueden representar una optimización importante en términos de rendimiento y consumo energético, su efectividad está inherentemente ligada a las prestaciones de la memoria, tanto la latencia como el ancho de banda, que determinan la velocidad a la que dichos datos pueden estar disponibles [2]. Por este motivo, uno de los retos de la arquitectura de computadores, común en el diseño tanto de procesadores escalares como vectoriales, es el desarrollo de técnicas que ayuden a reducir la latencia de memoria. A este respecto, las aplicaciones vectoriales presentan en general, una localidad de datos muy alta, por lo que soluciones que aprovechen bien esta localidad pueden proporcionar buenos resultados en este tipo de arquitecturas. Entre estas técnicas se encuentran la jerarquización de la memoria y el *prefetching* [3].

Otro de los retos, dependiente de la propia naturaleza de la investigación en este campo, es la ausencia de recursos físicos, temporales y económicos que permitan llevar a cabo la implementación de cada una de las diferentes propuestas arquitecturales. Sin embargo, es muy importante hacer una evaluación cuantitativa de las nuevas propuestas arquitecturales, antes de pasar al proceso de fabricación de los dispositivos. En consecuencia, la investigación en arquitectura de computadores se ha de realizar mediante la simulación, que es la forma en la que se trabajará en este proyecto.

1.2. Objetivos

El propósito principal de este trabajo es diseñar, implementar y analizar el rendimiento de una nueva propuesta de cache segregada, que diferencia particiones específicas para los datos escalares y vectoriales, en un procesador con soporte vectorial. La hipótesis de partida de esta propuesta es que tanto los patrones de acceso, como la localidad (espacial y temporal) de los datos escalares y vectoriales en las aplicaciones son muy distintos, por lo que utilizar estructuras de memoria específicas y adaptadas a cada caso puede proporcionar una ventaja significativa a este tipo de procesadores, tanto en rendimiento como en consumo energético. También se estudiará el impacto y la efectividad de emplear del *prefetching* sobre datos de la partición vectorial como técnica de reducción de la latencia de acceso a memoria de las instrucciones vectoriales.

Para evaluar la propuesta se utilizará, Cavatools [4], un simulador de la ISA RISC-V, en el que se ejecutarán diferentes benchmarks de aplicaciones vectoriales empleando distintas configuraciones de tamaño en los registros vectoriales.

1.3. Plan de trabajo

Los pasos necesarios para desarrollar el proyecto y alcanzar los objetivos previamente descritos son los siguientes:

1. Diseño de la propuesta de Cache Bicameral para un procesador vectorial RISC-V. Se definen las características, las estructuras y la funcionalidad de cada algoritmo asociado, tanto para la cache propuesta como para el resto del sistema de memoria que la completa.
2. Extensión del simulador Cavatools para implementar la Cache Bicameral. Se incorporan y modifican las clases necesarias de forma que se pueda simular la funcionalidad de esta cache.
3. Implementación de un modelado sencillo del controlador de memoria. Se extiende nuevamente el simulador para añadir la interacción de la cache con un sistema de memoria simplificado. En este punto, para tener en cuenta la temporalidad de cada operación y modelar latencias de manera precisa, se realizan las modificaciones necesarias del simulador para cambiar su modo de funcionamiento a dirigido por eventos.
4. Depuración y pruebas de corrección de los desarrollos realizados. Se realizan simulaciones con distintas configuraciones y benchmarks para analizar la precisión y correctitud de las funcionalidades implementadas comparando los resultados obtenidos con los esperados.
5. Simulación y análisis del rendimiento de la propuesta. Se simulan los benchmarks a evaluar para estudiar el rendimiento de la cache.

1.4. Estructura del documento

El contenido de este documento se organiza de la siguiente manera:

El *Capítulo 2* describe los conceptos y tecnologías que sirven de base para el desarrollo de este trabajo; la arquitectura de un procesador vectorial, el modelo de memoria, la extensión vectorial RISC-V y el simulador Cavatools. En él, también se ofrece una recopilación de trabajos relacionados que conforman el estado del arte de la investigación en los ámbitos de este trabajo; arquitectura vectorial y jerarquía de memoria.

El *Capítulo 3* presenta la propuesta realizada en este trabajo, la Cache Bicameral, explicando las características de su modelado, así como la interacción con el resto de la jerarquía de memoria.

En el *Capítulo 4* se recogen los detalles relativos a la implementación de la Cache Bicameral en el simulador Cavatools.

El *Capítulo 5* se realiza una evaluación de los resultados, detallando la metodología seguida y analizando las diferentes métricas obtenidas.

Por último, el Capítulo 6 sintetiza el trabajo realizado y las conclusiones obtenidas, además de proponer posibles vías de estudio futuro.

Capítulo 2. *Background*

En este capítulo se introducen y describen los conceptos y tecnologías sobre los que se basa este trabajo. También se recoge una colección de publicaciones previas relacionadas que ayudan a entender el contexto de la investigación en este ámbito.

2.1. Arquitectura de un procesador vectorial

El paralelismo a nivel de datos consiste en realizar una misma operación sobre varios elementos al mismo tiempo. La forma clásica de explotar este tipo de paralelismo es mediante una arquitectura vectorial [5].

Las arquitecturas vectoriales permiten que una única instrucción opere sobre un conjunto de datos o *vector*. Las instrucciones de este tipo, denominadas *instrucciones vectoriales*, son capaces de recuperar los datos referenciados por el vector de cada operando, colocarlos en registros, operar con ellos, y volver a ubicarlos en memoria en sus posiciones correspondientes. Por tanto, cada instrucción vectorial podría verse como un conjunto de operaciones escalares entre datos independientes, realizadas de manera simultánea.

El registro donde se colocan todos los datos de un vector para realizar las operaciones sobre ellos se denomina *registro vectorial*. Un procesador vectorial tiene varios registros vectoriales, y el tamaño de estos viene determinado por la longitud de vector máxima soportada por la arquitectura vectorial [5]. Es decir, si la arquitectura soporta una longitud máxima de vector de 2048 (como es el caso de la arquitectura ARM v8 [6]) el registro vectorial debe ser capaz de almacenar hasta 32 datos de 64 bits.

Además de los registros vectoriales, otro componente fundamental de un procesador vectorial son los *lanes*, que son los elementos de cómputo encargados de realizar las operaciones individuales sobre los elementos del vector [5]. Estos están formados por diferentes tipos de unidades funcionales enfocadas en realizar diferentes tipos de operaciones; aritmético-lógicas, con números enteros, con números en coma flotante, etc. Una operación vectorial puede realizarse únicamente sobre registros vectoriales o sobre una combinación de registro vectorial con registro escalar. Nótese que una arquitectura vectorial puede tener un número de *lanes* inferior a la longitud de vector máxima soportada. En tal caso, una instrucción vectorial requerirá una mayor cantidad de ciclos, en los que los *lanes* se utilizarán para operar sobre todos los elementos del vector.

Del mismo modo, también es necesaria una *unidad funcional de memoria* responsable de las lecturas y escrituras de vectores [5]. Este componente realiza la transferencia de los datos desde memoria a los registros vectoriales y viceversa. Los datos pertenecientes a un mismo vector no necesitan estar ubicados en posiciones adyacentes de memoria. La distancia de separación entre ellos se denomina *stride*. Cuando el *stride* es unitario, los datos son consecutivos. En caso contrario, los datos están *dispersos*. La unidad vectorial de memoria es capaz de reunir y dispersar los datos de cada vector desde y hacia memoria independientemente del *stride*.

2.2. Modelo de memoria

La jerarquía de memoria es la estructura en la que se organiza la memoria de los procesadores con el objetivo de optimizar tanto el coste como el rendimiento de los accesos aprovechando el principio de localidad y las diferentes tecnologías de fabricación de circuitos de memoria (SRAM, DRAM, etc.) [5].

Existen dos tipos de localidad; la temporal y la espacial. La localidad temporal define que, si un elemento acaba de ser referenciado, es probable que vuelva a serlo próximamente. La localidad espacial, por su parte, hace referencia a la tendencia a referenciar, en un futuro próximo, elementos cuyas direcciones están cerca del elemento que acaba de ser referenciado.

El diseño de la jerarquía de memoria trata de equilibrar las tres características principales de las memorias; el coste por bit, el tamaño y el tiempo medio de acceso. Así, a medida que se desciende en la jerarquía, o lo que es lo mismo, que aumenta la distancia con el procesador, las memorias son cada vez más lentas, grandes y baratas. De esta forma, y gracias al principio de localidad, el procesador trabajará fundamentalmente con los elementos que más cercanos a él se encuentran, es decir, los que conllevan un menor tiempo de acceso. Cuando un elemento referenciado no se encuentre en un nivel, se realizará una petición al nivel inferior. Idealmente, la frecuencia de accesos a elementos ubicados en niveles más bajos de la jerarquía será menor que la de elementos en niveles superiores.

Los registros del procesador constituyen el primer nivel de la jerarquía de memoria, y son el tipo de memoria más rápida, cara y pequeña de la jerarquía por ser locales a este. A continuación, se encuentran la memoria cache y la memoria principal.

2.2.1. Memoria cache

La cache es el nivel de la jerarquía de memoria que se encuentra entre la CPU y la memoria principal. Puede estar compuesto, a su vez, de varios dispositivos de memoria independientes denominados *niveles*, siendo el primer nivel, conocido como L1, el más cercano al procesador, y el último, conocido como LLC, del inglés *Last Level Cache*, el más alejado.

En un sistema de cache multinivel, es decir, con más de un nivel de cache, existen dos políticas de gestión de la información, una *inclusiva* y una *exclusiva* [7]. En un diseño inclusivo, cada nivel contiene un subconjunto de los datos del nivel inmediatamente inferior, lo que ayuda a simplificar el protocolo de coherencia, pero limita la capacidad efectiva de cache disponible [8, 7]. Por el contrario, en un diseño exclusivo, los datos presentes en un nivel de cache no pueden existir simultáneamente en otro. Un ejemplo de cache de nivel 2, o L2, inclusiva es la del procesador Intel Pentium 4 Willamette, mientras que uno de exclusiva se encuentra en el AMD Athlon Thunderbird 1141 [7].

Para obtener un dato, el procesador hace referencia a él en la cache. El tamaño del dato al que accede debe corresponder con una unidad direccionable. Generalmente, el acceso es a una *palabra*, la unidad natural de organización de memoria, si bien, dependiendo del sistema, se permiten otros tipos de direccionamiento, como a nivel de *media palabra* o de *byte*. Si el dato referenciado se encuentra en la cache, se considera un *acierto* o *hit,* y el procesador puede acceder a él. En caso contrario, se considera un *fallo* o *miss*, que conlleva una nueva referencia hacia el siguiente nivel de la jerarquía. Si el

sistema tiene un solo nivel de cache, el siguiente acceso se realiza a la memoria principal. En sistemas con más de un nivel de cache, se repite el procedimiento; solo en caso de fallo en el último nivel de cache, se accede a la memoria principal. Si el acceso al siguiente nivel de la jerarquía resulta en un acierto, se transfiere la información al nivel anterior. La unidad de transferencia entre niveles, generalmente, es un *bloque* o *línea*; un conjunto de palabras entre las que se encuentra el elemento referenciado por el procesador. Cada bloque o línea se identifica de manera única con una *etiqueta*, obtenida a partir de su dirección. A su vez, cada byte del bloque se identifica, de manera local a este, con un *desplazamiento*, que determina la posición del dato en el bloque [9].

Cuando un nuevo bloque se transfiere a una cache, debe ser ubicado en una línea de esta. Para determinar la línea concreta en la que se debe ubicar un nuevo bloque existen diferentes técnicas en función del tipo de correspondencia utilizada, es decir, del tipo de organización de la cache [10].

- *Correspondencia directa*. Cada bloque se mapea con una única línea de cache. El número de la línea *i* a la que se mapea un bloque se calcula como $i = j \bmod m$, siendo *j* el número del bloque y *m* el número de líneas de la cache. En este caso, los bits de la dirección de memoria se descomponen, de izquierda a derecha, en: *etiqueta*, *línea* y *desplazamiento*.

- *Totalmente asociativa*. Cada bloque puede mapearse con cualquier línea de cache. Por tanto, siempre que haya una línea vacía, esta se asignará en primer lugar. Solo en caso de que la cache esté llena, se elige una línea a expulsar para ubicar en su lugar el bloque referenciado. La posición concreta de dicha línea la determina el algoritmo de reemplazo utilizado, de modo que en cada acceso es necesario comparar en paralelo la etiqueta de todas las líneas de la cache con la de la dirección accedida. En este caso, los bits de la dirección de un bloque se descomponen, de izquierda a derecha, en: *etiqueta* y *desplazamiento*.

- *Asociativa por conjuntos*. La cache se divide en conjuntos o grupos de líneas. Todos los conjuntos tienen el mismo número de líneas o *vías*. Cada bloque se mapea con un único conjunto, pero puede ubicarse en cualquier línea de este. El número del conjunto *i* al que se mapea un bloque se calcula como $i = j \bmod m$, siendo *j* el número del bloque y *m* el número de conjuntos de la cache. La línea concreta del conjunto la determina el algoritmo de reemplazo utilizado, asignando siempre en primer lugar líneas vacías del conjunto, si las hay. Por tanto, en cada acceso es necesario comparar en paralelo la etiqueta de la dirección accedida con la de todas las líneas del conjunto correspondiente. En este caso, los bits de la dirección de un bloque se descomponen, de izquierda a derecha, en: *etiqueta*, *conjunto* y *desplazamiento*.

Las funciones de correspondencia que implican asociatividad requieren de una política de reemplazo para determinar qué línea debe ser *victimizada*, es decir, expulsada de la cache para poder ubicar en su lugar al nuevo bloque transferido. Existen multitud de algoritmos de reemplazo. Algunos de los más comunes son los siguientes:

- *LRU (Least Recently Used).* La línea victimizada es la menos recientemente utilizada. Este algoritmo trata de aprovechar al máximo la localidad temporal.

- *LFU (Least Frequently Used).* La línea victimizada es la que ha recibido menos referencias. Este algoritmo busca aprovechar patrones en la frecuencia de accesos.

- *FIFO (First In, First Out).* La línea victimizada es la más antigua, es decir, la que lleva más tiempo en la cache.

- *Random*: La línea a victimizar es escogida aleatoriamente.

Si la referencia a un bloque es una escritura, el bloque debe ser modificado. La técnica utilizada para actualizar un bloque en el resto de niveles de la jerarquía, de forma que el dato sea en todo momento coherente, depende de la política de escritura implementada en la cache. Existen dos opciones [5]:

- *Escritura directa (write-through).* La información se actualiza tanto en el bloque del nivel actual de la jerarquía de memoria como en el siguiente.

- *Post escritura (write-back).* La información se actualiza únicamente en el bloque del nivel actual de la jerarquía de memoria. Cada bloque tiene asociado un bit de modificación, o *dirty bit*, que vale 1 en caso de haber sido escrito al menos un dato en ese bloque. Solo cuando el bloque es reemplazado, es decir, expulsado de la cache, y su bit de modificación está activo, se actualiza la información en el siguiente nivel de la jerarquía de memoria.

Una técnica habitual de optimización de rendimiento en las caches para tratar de reducir la tasa de fallos es el *prefetching* o *precarga* [5]. Esta técnica consiste en la transferencia de uno o más bloques adicionales, además del referenciado, tras un fallo. Así, se busca aumentar la tasa de acierto en la cache y aprovechar el ancho de banda de memoria que no esté siendo utilizado para adelantar la transferencia de bloques que, probablemente, según el principio de localidad, vayan a ser referenciados próximamente. Si la predicción se cumple, las referencias a los bloques que han sido precargados, es decir, llevados a la cache antes de que el programa acceda a ellos, resultarán en aciertos en vez de en fallos, por lo que disminuirá el tiempo de acceso. En caso contrario, el impacto del *prefetching* puede llegar a ser negativo si supone el reemplazo de bloques útiles por bloques no útiles. Para determinar la efectividad de una técnica de *prefetching* existen diferentes métricas [3]:

- *Precisión (accuracy)*: fracción de datos precargados que resultan útiles, es decir, que acaban siendo referenciados. Una baja precisión puede suponer la *contaminación* de la cache (*cache pollution*), es decir, precargarla de datos innecesarios.

- *Puntualidad (timeliness)*: cualidad que relaciona el momento en el que el dato es precargado respecto del momento en el que va a ser necesitado. Si

el dato no llega con la suficiente puntualidad, pueden producirse dos efectos negativos:

- *Prefetching temprano*: el dato llega mucho antes de ser necesario. Esto puede provocar reemplazos en la cache de datos que fueran a ser referenciados antes del dato precargado. En caches de baja asociatividad, el dato precargado también podría llegar a ser reemplazado antes de ser utilizado. Por ello, un *prefetching* temprano puede aumentar tanto los fallos de cache como el tráfico de memoria.
- *Prefetching tardío*: el dato llega tras haber sido referenciado. Si bien esta falta de puntualidad no oculta los fallos de cache, ayuda a reducir la latencia de estos.

– *Cobertura (coverage)*: fracción de fallos evitados por el adelantamiento del dato. Su valor puede verse reducido tanto por la baja precisión como por la baja puntualidad.

2.2.2. Memoria principal

La memoria principal es el elemento fundamental de memoria de un procesador. Generalmente, está formada por un conjunto de chips DRAM, a diferencia de las cache, que habitualmente emplean tecnología SRAM.

La tecnología de memoria de acceso aleatorio o RAM, del inglés *Random-Access Memory*, es un tipo de memoria semiconductora que permite tanto la lectura como la escritura. Esta memoria, además, es volátil, por lo que pierde la información almacenada en sus celdas si se interrumpe la corriente eléctrica [10]. Una celda de memoria SRAM, o RAM *estática*, mantiene la información almacenada sin necesidad de refresco empleando seis transistores por bit [5]. Por el contrario, una celda de memoria DRAM, o RAM *dinámica*, almacena cada bit de información en un condensador [9]. Debido a que la carga de los condensadores se disipa de manera gradual, todas las celdas que componen un chip de memoria DRAM requieren un refresco de manera periódica.

La tecnología SRAM emplea celdas más grandes y rápidas que la DRAM, por lo que, aunque su coste por bit es más elevado, reduce el tiempo de acceso. Por el contrario, la tecnología DRAM, al emplear celdas más lentas, pero más pequeñas, permite mayor densidad con menor coste por bit y, por ende, resulta más adecuada para memorias de mayor capacidad.

Los chips DRAM se organizan físicamente en tarjetas de memoria denominadas módulos. Tradicionalmente, se emplean módulos DIMM, del inglés *Dual-Inline Memory Module*, que contienen dos filas de varios chips cada uno. Un tipo de optimización que habitualmente se emplea en los chips de memoria DRAM es la tecnología DDR, del inglés *double data rate*; que permite la transferencia de los datos tanto en el flanco de subida del reloj como en el de bajada, duplicando así la tasa de transmisión [10].

Internamente, la memoria DRAM se organiza en *bancos*. Cada banco de memoria contiene un array bidimensional de celdas que se accede por *filas* [11] [12]. El acceso a una fila de un banco, o *activación* de la fila, implica que esta se transfiera completamente al *buffer de fila abierta*. Este buffer, único por banco, permite almacenar el contenido de

la fila abierta para reducir la latencia en futuros accesos a elementos de esta. Dado que las celdas de memoria DRAM están formadas por condensadores, las lecturas son destructivas, es decir, provocan la pérdida de la carga en estos [9]. Además, en un banco solo puede haber una única fila abierta al mismo tiempo, que es en la que se pueden efectuar operaciones de lectura y escritura. Por ello, para sustituir una fila abierta por otra, primero se debe *cerrar* la fila abierta, lo que implica volver a escribir el contenido almacenado en el buffer en la fila correspondiente del banco.

En la *Figura 1* se recoge, de manera esquemática, la relación de estados de una fila en función de las operaciones que se realizan sobre ella. La operación de lectura o activación de una fila se conoce como RAS, del inglés *Row Access Strobe* [13]. Una vez abierta, para leer o escribir un bit de esta es necesario acceder a la columna correspondiente. Esta operación se conoce como CAS (*Column Access Strobe*). Por último, la operación de cierre de fila o recargado, se denomina PRE, del inglés *Precharge*.

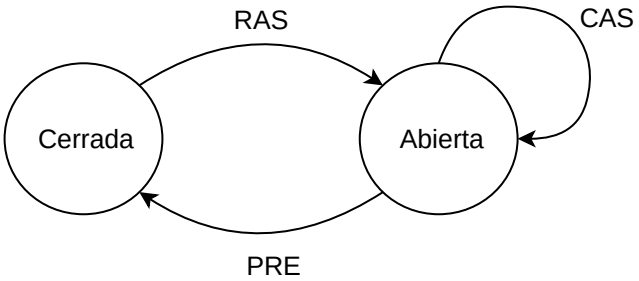

Figura 1. *Diagrama de estados de la fila de un banco de memoria DRAM. Fuente: [12].*

El controlador de memoria es el elemento que se encarga de atender a las peticiones del procesador realizando las operaciones de acceso a memoria DRAM correspondientes [11]. Para ello, almacena las peticiones pendientes en una cola o buffer hasta que puedan ser planificadas. Esta cola puede estar segmentada para organizar las peticiones en función del banco de memoria al que hacen referencia [11]. Los datos leídos de memoria a la espera de ser transferidos, así como los datos recibidos para ser escritos en memoria, también se conservan en sendas colas hasta que su instrucción correspondiente se haya procesado por completo. El controlador de memoria gestiona el orden de planificación de las peticiones en función de la política de prioridad que implemente, respetando siempre las restricciones temporales específicas de las celdas DRAM. También es el encargado de llevar a cabo los refrescos periódicos de estas [5]. La comunicación con el resto del sistema (la cache y el procesador), se realiza mediante buses de memoria dedicados [10].

2.3. Extensión vectorial de RISC-V

RISC-V es una ISA de código abierto surgida en el año 2010 en la Universidad de California en Berkley con el objetivo de convertirse en un conjunto de instrucciones estándar y universal [14]. El diseño de RISC-V se basa en los principios de la arquitectura RISC, del inglés *Reduced Instruction Set Computer*, que busca simplificar el hardware de los computadores definiendo instrucciones lo más sencillas posibles [14] [15].

La extensión vectorial de RISC-V, también conocida como RVV o RISC-V V, añade el conjunto de instrucciones vectoriales a la ISA. Esta extensión es independiente de la longitud de vector, lo que permite la portabilidad de cualquier código definido con estas instrucciones entre cualquier procesador vectorial que las implemente [14].

RVV define 32 registros vectoriales del mismo tamaño, determinado por la longitud de vector, o *VLEN*, y 7 registros de control y estado, o *CSR* [16]. Los vectores se dividen en elementos, que tienen un tamaño fijado por el parámetro denominado *ELEN*. Por tanto, el número de elementos de un vector viene definido por *VLEN / ELEN*. Ambos parámetros son constantes determinadas por la implementación que corresponden a un valor de número de bits potencia de 2. En el caso de *ELEN*, el valor debe encontrarse en el intervalo *[8, VLEN]*.

Dos de los registros *CSR* de la extensión se utilizan como operandos implícitos de las instrucciones vectoriales; los registros *vtype* y *vl*. Los valores de ambos registros se pueden establecer con las instrucciones *vset{i}vl{i}*.

El registro *vtype* define el tipo de vector sobre el que se opera, o lo que es lo mismo, determina la organización de los elementos en cada registro vectorial. De sus cinco campos, destacan *vsew* y *vlmul*. Por un lado, *vsew*, del inglés *Selected Element Width*, define la longitud estándar en número de bits de los elementos sobre los que se opera. Este valor no puede ser superior a *ELEN*. Por su parte, *vlmul*, del inglés *Length Multiplier*, define el número de registros vectoriales que hay en un *grupo*. El concepto grupo hace referencia a la cantidad de registros vectoriales que se emplean de manera conjunta como un único operando en una instrucción vectorial.

El registro *vl* define el número de elementos totales sobre los que opera con una instrucción vectorial. Este valor debe ser, a lo sumo, *vlmax = VLEN / vsew * lmul*, es decir, el número de elementos de longitud estándar *vsew* en un registro vectorial multiplicado por el número de registros vectoriales que conforman el grupo [17]. Sin embargo, *vl* puede ser inferior, pues una instrucción vectorial puede no utilizar todos los elementos del grupo. Al conjunto de elementos restantes se le denomina *cola*. Tras una instrucción vectorial, en función de la política aplicada, la cola puede quedar intacta o bien adquirir el valor 1 en todos los bits de sus elementos.

En la *Figura 2* se recoge la representación gráfica de dos instrucciones vectoriales de RVV. La primera instrucción *vsetivli* establece en el registro *x10* el valor de *vl* a 3 elementos de tamaño estándar *vsew = 32 bits*. Además, define un grupo de un único registro vectorial (*vlmul)*. En el ejemplo, la longitud de vector *VLEN* es de 128 bits, por lo que hay 4 elementos de tamaño estándar (*vsew*) en cada registro vectorial, que en este caso coincide con el valor de *vlmax*. La segunda instrucción *vadd.vv* define una operación de suma *ADD* de los registros vectoriales *v4* y *v5*, almacenando el resultado en *v6*. Esta operación se realiza elemento a elemento sobre los 3 elementos de longitud estándar que conforman *vl*, siendo el último elemento de cada registro vectorial la cola.

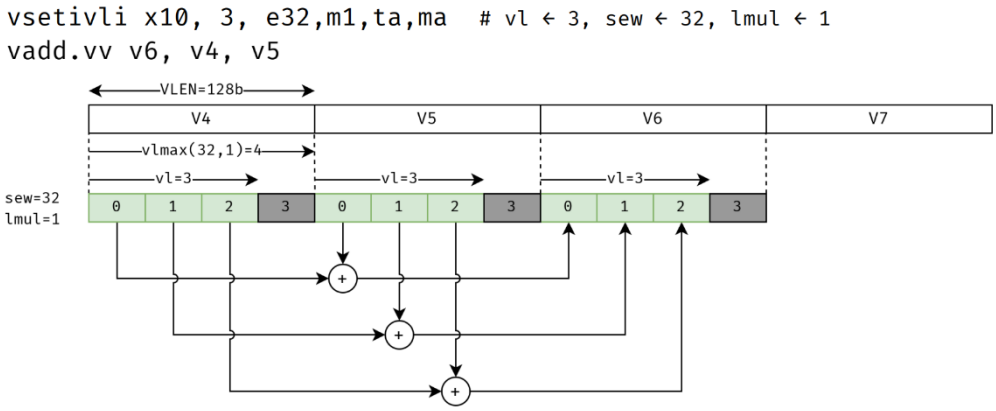

Figura 2. *Ejemplo gráfico de cómo operan las instrucciones vectoriales de RISC-V V. Fuente: [17].*

2.4. Simulador Cavatools

Cavatools[1] es un simulador de código abierto, desarrollado para sistemas Linux x86, que está orientado al diseño de extensiones personalizadas de RISC-V y la evaluación de su rendimiento en aplicaciones reales [18] [4].

Este simulador ejecuta aplicaciones desarrolladas con instrucciones de la ISA RISC-V, incluyendo algunas de sus extensiones, como la RVV. Cavatools también proporciona una interfaz Linux de llamadas al sistema en modo usuario para el binario de la aplicación, por lo que, desde el punto de vista del usuario, esta parece estar corriendo directamente como un proceso ordinario de Linux que utilizara las llamadas a sistema nativas. El resultado es que cualquier aplicación compilada con las herramientas estándar de RISC-V puede ser ejecutada en el simulador.

La metodología de evaluación de arquitecturas que emplea Cavatools, basada en la descomposición, es la clave de su rapidez. Esta técnica se basa en descomponer cada simulación en diferentes simulaciones más simples, lo que permite analizar la ejecución de un gran número de instrucciones en el mismo periodo de tiempo [18].

Cavatools está formado por tres elementos; *uspike*, *caveat* y *erised* [4]. El primero es el intérprete de RISC-V, que es el encargado de la simulación puramente funcional de las instrucciones. La codificación en bits de estas, así como su semántica, se extraen automáticamente, mediante scripts de Python, del repositorio GitHub oficial de la ISA. *Caveat* es el simulador de rendimiento. Este permite modelar un chip multicore con pipeline *single-issue* en orden y caches privadas de nivel 1 tanto de instrucciones como de datos. Por último, *erised* es un visor de rendimiento no intrusivo que representa en tiempo real la frecuencia de ejecución de las diferentes partes de la aplicación empleando diferentes códigos de color. Concretamente muestra, para cada instrucción, su código en ensamblador y sus estadísticas de frecuencia y CPI asociadas. También incorpora funcionalidades de zoom y desplazamiento por la ventana para examinar los distintos detalles.

[1] Enlace al repositorio de Cavatools: https://github.com/phaa-eu/cavatools.

2.5. Trabajos relacionados

En la literatura se encuentran numerosos ejemplos de investigación llevados a cabo sobre arquitecturas vectoriales y jerarquía de memoria. A continuación, se resumen algunos cuyas líneas de trabajo se acercan a la descrita en este documento.

Lee *et al.* [19] proponen un nuevo controlador de memoria orientado a mejorar el rendimiento de las operaciones de *prefetching*. Este controlador realiza una estimación de la utilidad de cada petición de *prefetch* en base a su precisión y puntualidad en el programa para decidir, de manera adaptativa, si debe ser priorizada o desestimada. Con ello consiguen no solo aumentar el rendimiento de las aplicaciones en el sistema, sino también aprovechar más eficientemente el ancho de banda de memoria.

Batten *et al.* [20] presentan la VRU, una unidad desacoplada de rellenado vectorial que actúa como *prefetcher* de las líneas de cache. Esta unidad sondea la cola de las instrucciones vectoriales de memoria con antelación y emite, de forma anticipada, peticiones a la memoria principal para rellenar las líneas asociadas a dichas referencias. De esta forma, para cuando la unidad de ejecución vectorial ejecuta cada instrucción, sus datos ya están disponibles en la cache. La VRU demuestra alcanzar mayor rendimiento a la vez que reduce la cantidad de recursos hardware necesarios en comparación a los sistemas desacoplados tradicionales.

En el trabajo de Espasa y Valero [21] se desarrolla una nueva cache dedicada para los registros vectoriales que aborda el problema del *spilling*. El *spilling* de registros se define como el tráfico de memoria adicional generado para preservar el contenido de los registros únicamente con el objetivo de liberarlos para lidiar con la escasez de estos. Esta cache a nivel de registro tiene capacidad para almacenar varios vectores completos victimizados de los registros únicamente por instrucciones de escritura. Su evaluación ofrece notables mejoras en la reducción del tráfico por *spilling*; de 5% a 20% para la versión *write-through* y de 10% a 50% para *write-back*.

En la propuesta de Musa *et al.* [22] se presenta una cache vectorial que incorpora registros de gestión del estado de los fallos o MSHR (*Miss Status Handling Registers)* y un mecanismo de *prefetch*. Dicha cache está dividida en 32 sub-caches que emplean una organización asociativa por conjuntos de 2 vías, política de escritura directa y algoritmo de reemplazo LRU. El registro MSHR de cada sub-cache almacena la dirección de memoria de cada petición de lectura *en vuelo* enviada a memoria para evitar que diferentes instrucciones realicen peticiones duplicadas de los mismos datos. Con ello, logran mejorar el rendimiento desde un 5% hasta un 45%, dependiendo del sistema. Para el mecanismo de *prefetch* se emplea una solución software basada en la inserción de instrucciones de *prefetch* en el propio archivo de trazas de la simulación para aprovechar, bien las latencias no ocultas de peticiones a memoria, o bien el reúso de los datos *prefetcheados* por múltiples instrucciones. Con este, en los sistemas evaluados, se consiguen mejoras del rendimiento del 20% al 60%.

Ramírez *et al.* [23] contribuyen a ampliar las herramientas de investigación en arquitecturas vectoriales disponibles con dos aportaciones diferentes; una extensión del simulador gem5 que da soporte a las instrucciones vectoriales de RISC-V y una suite de aplicaciones vectoriales que permiten evaluar diferentes componentes de la arquitectura vectorial.

Hsu y Smith [24] estudian el impacto en el rendimiento de incluir un sistema interno de cache en la propia memoria principal DRAM de un supercomputador vectorial. En su evaluación comparan diferentes métodos de intercalado de memoria, resultado de diferentes distribuciones de los campos de la dirección, y dos tipos de cache interna en la DRAM; una cache de una única fila larga y una cache de varias filas cortas. Sus resultados demuestran que el intercalado resulta ser generalmente beneficioso, si bien el método exacto debe optimizarse para equilibrar el aprovechamiento de la localidad espacial y la uniformidad de acceso a los bancos. Para sistemas monoprocesador, ambos tipos de cache proporcionan buenos resultados siempre que se aplique algún tipo de intercalado, mientras que para sistemas multiprocesador, únicamente la cache de múltiples líneas mejora el rendimiento.

En el trabajo de Fu y Patel [25] se compara el rendimiento de dos métodos de *prefetching* en una cache vectorial de un sistema multiprocesador, adaptando el tamaño accedido por una lectura de cache al tipo de dato referenciado. La evaluación demuestra que el método simple de *prefetching* secuencial, que únicamente realiza *prefetch* en referencias escalares o vectoriales con *stride* menor al bloque de cache, obtiene mejoras en el *speedup* general del sistema frente a la ausencia de *prefetching*, contribuyendo a reducir el impacto negativo de los accesos vectoriales de *stride* largo, a pesar de no aumentar significativamente la tasa de aciertos en la propia cache. Por su parte, el método de *prefetching* con *stride*, que emplea el mismo *stride* de la referencia para el *prefetch*, sí consigue mejorar el rendimiento de la cache frente al método secuencial. En la mayoría de casos, además, proporciona un *speedup* más alto.

Rothman y Smith [26] reintroducen el concepto de *sector cache* o cache sectorizada, el modelo que empleó el primer computador comercial con memoria cache. Este tipo de cache está compuesta de conjuntos de *sectores*, cada uno con una etiqueta asociada, que se dividen en *sub-sectores*, cada cual con sus bits de validez y modificación. A través de un análisis del rendimiento mediante simulación por trazas de este tipo de caches, totalmente asociativas y con distintos tamaños y grados de sectorización, sobre múltiples cargas de trabajo, concluyen que, a pesar de que, en promedio, un 72% de los subsectores no llegan a ser a ser referenciados, el modelo de cache sectorizada ofrece mejoras de rendimiento significativas frente al modelo habitual en sistemas de cache multinivel con un tamaño pequeño en el primer nivel.

La propuesta de la Cache Bicameral desarrollada en este trabajo emplea la idea de la sectorización analizada por Rothman y Smith [26] únicamente para la partición de las referencias vectoriales. Como en esta, cada línea está dedicada a almacenar los elementos consecutivos de un mismo vector, se espera aprovechar las ventajas de la sectorización a la vez que evitar la despoblación de dichas filas. Esta cache incorpora una opción de *prefetching* secuencial como el evaluado por Fu y Patel [25], pero únicamente sobre los elementos de las filas vectoriales, es decir, similar al que plantean Batten *et al.* [20], con el objetivo de aumentar la tasa de acierto a la vez que se evitan tanto la contaminación de la cache como los reemplazos indeseados. Este *prefetching* incorpora una funcionalidad similar a la del registro MSHR presentado por Musa *et al*. [22] para evitar la duplicidad de peticiones sobre los mismos datos. La evaluación de dicha propuesta se apoya en algunos de los benchmarks vectoriales de la contribución de Ramírez *et al.* [23].

Capítulo 3. Diseño y modelado de la Cache Bicameral

En este capítulo se desarrolla la propuesta de la Cache Bicameral, describiendo sus características y los detalles modelados para su evaluación, así como los del resto de la jerarquía de memoria con la que interactúa.

3.1. Objetivo general

La propuesta de la Cache Bicameral es una nueva propuesta de memoria cache de datos para procesadores vectoriales que se basa en aunar dos caches con distintas geometrías, para tratar de adaptarse a los diferentes requisitos sobre la jerarquía de memoria impuestos por las instrucciones escalares y vectoriales. Así, la idea general consiste en capturar la localidad temporal inherente al cómputo vectorial, evitando las potenciales interferencias que puede introducir el cómputo escalar. Gracias a la largura de las líneas de la cache vectorial, así como a la funcionalidad adicional de rellenado de estas, también se busca explotar la localidad espacial de los datos vectoriales.

Para ello, se propone una memoria cache escalar convencional y una cache vectorial totalmente asociativa compuesta por líneas sensiblemente más largas. Las líneas de ambas caches están *sectorizadas,* es decir, divididas en *sectores*, siguiendo un diseño similar al de la *Sector Cache* de Rothman y Smith [26], previamente mencionada en el *apartado 2.5.* Un sector es la unidad mínima de transferencia de datos entre la memoria principal y la cache. El tamaño de cada sector es siempre el mismo y coincide con el tamaño de las líneas de la cache escalar. Las líneas de la cache vectorial, al ser más largas, están formadas por varios sectores, a diferencia de las de la escalar, que solo contiene un único sector cada una. De este modo, si bien cada línea de cache tiene asociada únicamente una etiqueta que actúa de identificador, como es habitual, cada uno de sus sectores tiene asociados dos bits adicionales; un bit de validez *(v)* y un bit de modificación o *dirty bit (d).* En la *Figura 3* se representa la estructura de las líneas y los sectores en estas caches.

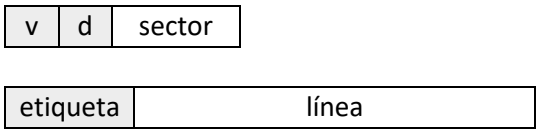

***Figura 3**. Representación de la estructura de una línea y un sector de la Cache Bicameral.*

Asimismo, se propone una política de *prefetching* sencilla con el objetivo de aprovechar la naturaleza *streaming* del cómputo vectorial, rellenando la cache anticipadamente y mejorando el ancho de banda ofrecido por el controlador de memoria. Las siguientes secciones explican en detalle los elementos que conforman la Cache Bicameral, utilizando los parámetros y geometría escogidos para su posterior evaluación. No obstante, los conceptos generales aplican con independencia de los números concretos.

3.2. Cache Bicameral

La Cache Bicameral está compuesta de dos tipos de cache; la cache escalar y la cache vectorial. La estructura y características de cada una se describen a continuación.

3.2.1. Cache escalar

La cache escalar, o SC, por sus siglas en inglés, almacena los datos referenciados por las instrucciones escalares. Es una cache asociativa por conjuntos que emplea el algoritmo LRU como política de reemplazo. Posee 256 conjuntos de 4 líneas cada uno. Las líneas tienen una longitud de 64B, por lo que cada una almacena un único sector de datos de 64 B. De este modo, su capacidad total es de 64 KB. En la *Figura 4* se recoge la representación gráfica de su organización.

Esta cache emplea la política de actualización de post escritura. Por ello, tiene un búfer de escritura o *write buffer (WB)*, donde almacena las líneas expulsadas cuyos sectores han sido modificados. El objetivo de este buffer es agilizar el rendimiento, tratando de no bloquear al procesador a la espera de que las escrituras en memoria de dichos sectores se completen. El *write buffer*, también representado en la *Figura 4*, tiene una capacidad máxima de 8 líneas.

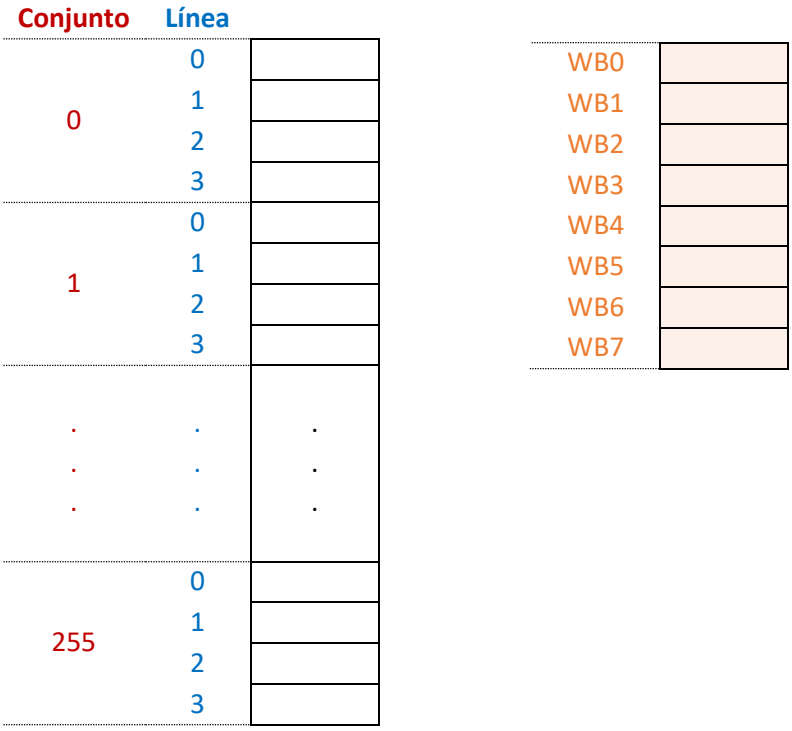

***Figura 4**. Representación de la cache escalar a la izquierda, y su write-buffer a la derecha.*

Dada la configuración de la cache escalar, su dirección de 32 bits se descompone en tres campos, como se muestra en la *Figura 5*. Los 6 bits menos significativos $(log_2 64)$ corresponden al *desplazamiento (d)*, es decir, sirven para identificar el dato referenciado dentro del sector. Los 8 siguientes $(log_2 256)$ identifican el *conjunto (j)* al que pertenece el sector, y los 18 restantes corresponden a la *etiqueta (e)*.

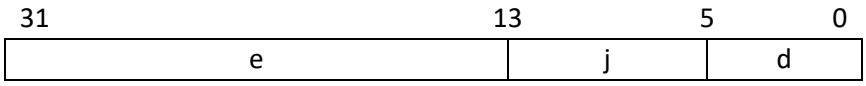

***Figura 5**. Campos de la dirección de la cache escalar.*

3.2.2. Cache vectorial

La cache vectorial (VC) almacena los datos referenciados por las instrucciones vectoriales. Esta cache es totalmente asociativa. Sus líneas tienen una capacidad de 1024B, por lo que cada una almacena 16 sectores de datos de 64 B. Al igual que la cache escalar, emplea el algoritmo LRU para el reemplazo, la política de actualización de post escritura y cuenta con un *write buffer* de un máximo de 8 líneas. En él se almacenan las líneas victimizadas que contienen al menos un sector modificado que debe ser actualizado en memoria principal. No obstante, como novedad, este *write buffer* está incorporado en la propia cache, de forma que sus líneas no ocupadas se comportan como líneas de cache adicionales. Esto implica que la cache vectorial tiene un total de 72 líneas, 8 de las cuales pueden estar dedicadas al *write buffer*. Así, cuando el *write buffer* esté lleno, es decir, haya ocho líneas asignadas a este, la capacidad efectiva de la cache vectorial será la misma que la de la cache escalar, 64 KB, mientras que, cuando no lo esté, esas líneas podrán ser utilizadas como líneas de cache habituales. Esta decisión de diseño está motivada por la largura de las líneas de la cache vectorial, lo que conllevaría el desperdicio de gran capacidad de almacenamiento si el *write buffer* fuera independiente y dedicado como el de la cache escalar, en vez de dinámico e integrado.

Para alcanzar la misma capacidad que la cache escalar, la cache vectorial emplea 64 líneas de datos. Esto significa que el tamaño efectivo de la cache varía dinámicamente entre 64 KB y 72 KB, en función de la ocupación del *write buffer*. En la *Figura 6* se representa la organización de la cache vectorial.

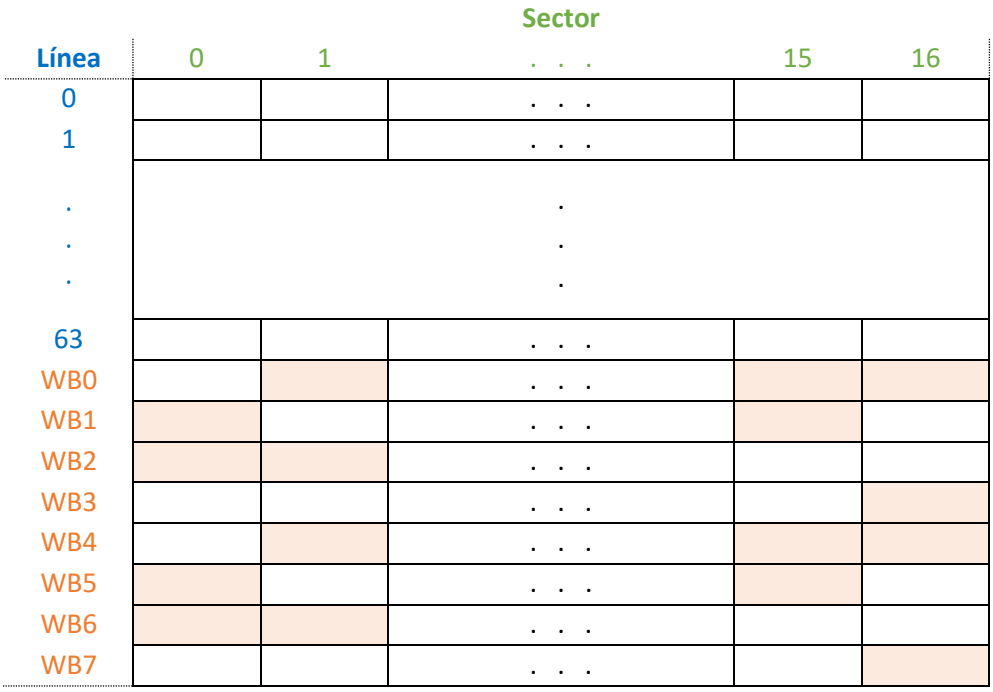

***Figura 6.** Representación de la cache vectorial. Los sectores resaltados de las líneas de write buffer tendrían que ser escritos a memoria principal (válidos y modificados).*

A partir de la configuración de la cache vectorial, su dirección de 32 bits se descompone en los campos representados en la *Figura 7*. Igual que en la cache escalar, sus 6 bits menos significativos *($log_2 64$)* corresponden al *desplazamiento (d)*. Los 4 siguientes *($log_2 16$)* identifican el *sector (s)* dentro de la fila, por lo que los 22 más significativos corresponden a la *etiqueta (e)*.

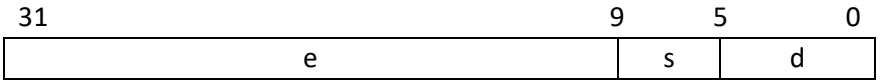

***Figura 7**. Campos de la dirección de la cache vectorial.*

Ambas caches son mutuamente exclusivas, por lo que un sector no puede encontrarse en las dos a la vez. Esto implica que, tras un fallo en la cache correspondiente al tipo de instrucción que lo referencia, se debe acudir a la cache contraria, realizando una referencia *cruzada*, antes de realizar la petición a memoria principal. De este modo, cada cache hace las veces de una *pseudo cache de nivel 2* para la otra, pero únicamente se encarga de la búsqueda del dato. Si la referencia cruzada resulta en un fallo, es la cache original, y no la cruzada, la que realiza la petición de memoria para ubicar el dato en una de sus líneas. Cuando la referencia cruzada desde la cache vectorial resulta en un acierto, el sector se migra; deja de estar en la cache escalar y se incorpora a la vectorial. Esta migración, que no se produce en sentido contrario, pretende priorizar la presencia de los datos de los vectores en la cache vectorial, con el objetivo de favorecer el rendimiento de las instrucciones vectoriales. Así, aunque una instrucción escalar referencie un dato vectorial, este se mantendrá en la cache vectorial si ya residía en ella. Si no, se incorporará a la cache escalar, pero será migrado a la vectorial tan pronto como una instrucción vectorial lo referencie.

Cabe destacar que una única instrucción vectorial de acceso a memoria genera múltiples referencias, tantas como elementos tenga su vector, por lo que el número depende del tamaño arquitectural establecido para este. Cada una de estas referencias realiza un acceso a memoria independiente generado en el mismo ciclo de manera simultánea. Para modelar este comportamiento, por simplicidad, se supone una única referencia a la cache vectorial por cada sector.

3.3. Resto de la jerarquía de memoria

Para modelar con precisión el comportamiento de la Cache Bicameral se ha de tener en cuenta la interacción con el resto de la jerarquía de memoria. Con este objetivo, se ha realizado también un modelado simplificado de la memoria principal y del controlador de memoria.

3.3.1. Memoria principal

La memoria principal modelada es una tecnología de tipo DRAM. Cuenta con 8 bancos, cada uno de los cuales puede tener una única fila abierta al mismo tiempo. En cada banco, hay 32.768 filas y 256 columnas. La distribución de los campos de su dirección de 32 bits sigue un patrón *Row-Bank-Column* estándar, que se muestra en la

Figura 8. Los 6 bits menos significativos *(log_2 64)* determinan el *desplazamiento (d)* en el sector de 64 B. Los 8 siguientes *(log_2 256)* indexan la *columna (c)*. Luego, están los 3 bits *(log_2 8)* para en *banco (b*). Los restantes 15 bits *(log_2 32768)* seleccionan la *fila (f)*. La capacidad total de la memoria principal es de 4 GB.

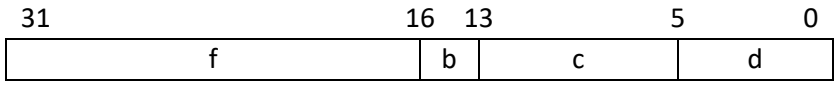

Figura 8. *Campos de la dirección de la memoria principal.*

El controlador de memoria es el responsable de gestionar las operaciones que se realizan en la memoria principal. Tiene una cola de peticiones pendientes que atiende en orden de llegada. En ella se encolan las peticiones de búsqueda o *lookup* y escritura (*write-back*) de los datos que solicitan las caches. La comunicación entre la Cache Bicameral y el controlador de memoria se ha modelado con dos buses de transferencia; uno en cada dirección. De este modo, las caches pueden enviar peticiones de *lookup* o escritura a memoria a la vez que reciben datos de esta. El tamaño definido para estos buses es de 256 bits, que constituye un tamaño adecuado para una arquitectura vectorial.

Para simplificar el modelado, se ha determinado que sobre la memoria principal solo se realiza una única operación a la vez. Además, se ha obviado la operación de refresco de las celdas DRAM y otros comandos complejos. Las únicas operaciones tenidas en cuenta para el modelado son las explicadas en el *Capítulo 2* y representadas en la *Figura 1*.

3.3.2. Prefetching

Además de gestionar las operaciones de *lookup* y *write-back* sobre la memoria principal, el controlador de memoria también se encarga del *prefetching*. Esta funcionalidad permite planificar *lookups* adicionales de sectores pertenecientes a líneas de la cache vectorial con el objetivo de poblarlas con antelación para intentar reducir la tasa de fallos, minimizando así las latencias de espera del procesador en las operaciones vectoriales.

Para no perjudicar el rendimiento interfiriendo en los accesos a memoria, la planificación del *prefetching* debe realizarse únicamente cuando el core no esté bloqueado esperando una respuesta de memoria principal. De este modo, el controlador de memoria planifica cada operación de *prefetch* sobre un único sector a la vez y exclusivamente cuando la cola de peticiones de memoria está vacía, es decir, cuando no hay ninguna instrucción de memoria en vuelo dependiente de la DRAM.

Para tratar de aprovechar el principio de localidad espacial, el *prefetching* se realiza sobre la misma línea de la última referencia a memoria realizada por la cache vectorial. Concretamente, según el algoritmo simple modelado con patrón secuencial, el *prefetch* se realiza en el siguiente sector inválido por la derecha, si lo hubiera, del sector que fue referenciado en dicha instrucción. Al finalizar el *prefetching*, se resuelve también, si la hubiera, la petición de *lookup* sobre el mismo sector que estuviera encolada como pendiente. Nótese que este *prefetch* en ningún momento conlleva reemplazos de datos ya presentes en la cache vectorial, es decir, no produce contaminación, pues su función es únicamente rellenar líneas ya existentes con sectores consecutivos que falten.

Capítulo 4. Implementación

La implementación de la Cache Bicameral y sus componentes asociados se ha realizado en el simulador Cavatools. En la *Figura 9* se recoge la relación simplificada de las clases que han sido modificadas, representadas en rosa, y creadas, representadas en azul, para llevar a cabo dicha implementación.

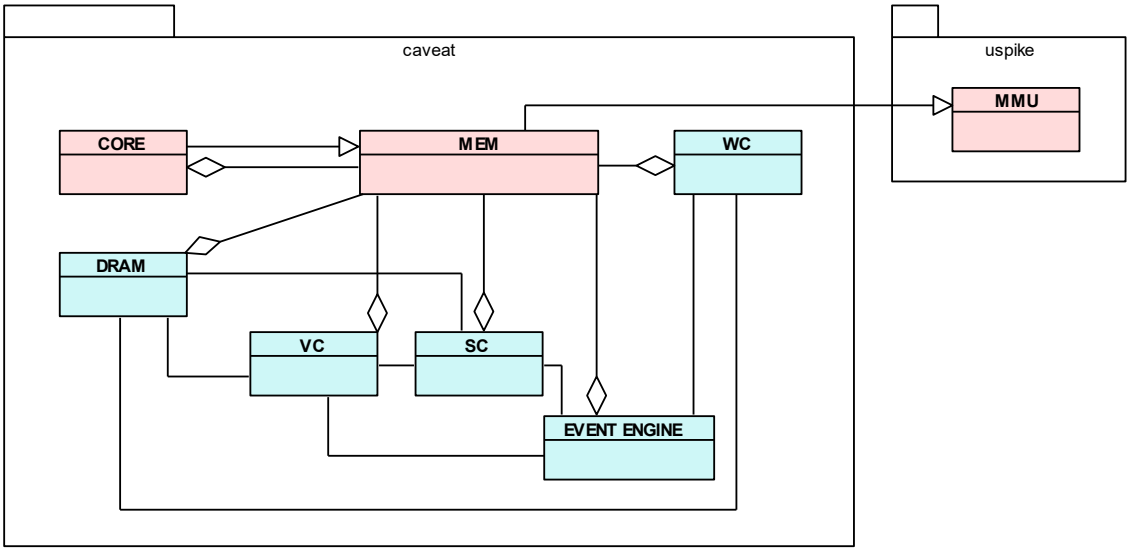

Figura 9. *Diagrama de clases simplificado de Cavatools.*

Las siguientes secciones, que se apoyan en el diagrama de clases de dicha figura, describen las principales funcionalidades añadidas al simulador para permitir la evaluación de la Cache Bicameral.

4.1. Etiquetado de instrucciones escalares y vectoriales

La clase *MMU* (*Memory Managenment Unit*) de *uspike* se ha modificado para incorporar la funcionalidad de añadir un *flag* de vectorización a los códigos de operación (*opcodes*) de las instrucciones vectoriales de RISC-V. Este *flag* permite identificar las instrucciones vectoriales y diferenciarlas de las escalares, por lo que es clave para determinar la cache que debe referenciar cada una.

Concretamente, dicho *flag* es utilizado por la clase *MEM* de *caveat* para planificar las referencias de cada instrucción de memoria a la cache correspondiente. Esta clase es la encargada de gestionar el principio y el fin de todas las instrucciones de memoria. Además, filtra el número de referencias producidas por la misma instrucción vectorial de forma que en el mismo ciclo, solo se planifique, a lo sumo, una única petición por sector. Una instrucción vectorial de memoria se da por finalizada únicamente cuando todas sus referencias asociadas han finalizado. La clase *MEM* también se encarga de la planificación de la siguiente instrucción tras la finalización de cada instrucción de memoria.

4.2. Motor de eventos

Cavatools es un simulador dirigido por ejecución, lo cual dificulta el adecuado modelado de las restricciones temporales que conlleva la ejecución de cada instrucción, particularmente en lo que al acceso a memoria se refiere. Para ello, se ha modificado el simulador para convertirlo en dirigido por eventos. Este modo de simulación aporta mayor precisión al incorporar un reloj que gobierna la planificación de las diferentes operaciones en función del número de ciclos que requieren (*penalización*). Esta funcionalidad se ha implementado con la clase *EVENT ENGINE* en *caveat*.

Por su parte, la clase *CORE* de *caveat* define el propio core del sistema, que se encarga de planificar las nuevas instrucciones, teniendo en cuenta sus latencias, siempre que no se acabe de ejecutar una instrucción de memoria, y gestionar el avance del reloj de dicho motor de eventos.

El motor de eventos se basa en un *timing wheel* o rueda temporal, implementado como un buffer circular en el que cada posición determina un ciclo distinto. La lista de eventos planificados para un mismo ciclo, si bien se ejecuta de manera secuencial en el simulador, se considera paralela a efectos de la simulación, ya que todos ocurren en la misma franja de tiempo. Únicamente cuando ha ejecutado todos los eventos planificados para un ciclo, el core avanza el reloj, incrementando su contador, y procede a ejecutar los planificados para el siguiente.

4.3. Caches escalar y vectorial

La implementación de la Cache Bicameral se ha llevado a cabo mediante la creación de las clases *VECTOR CACHE (VC)* para la cache vectorial, y *SCALAR CACHE (SC)* para la escalar.

Ambas caches contienen una estructura de datos que representa la organización de su almacenamiento. En estas estructuras se almacenan únicamente las etiquetas de las líneas y los bits de validez y modificación de los sectores; no los datos de estos como tal. Para modelar y evaluar el rendimiento no es necesario realizar la transferencia explícita de datos en la simulación, por lo que solo se han tenido en cuenta las penalizaciones en número de ciclos y las modificaciones en el estado de los componentes del sistema que conlleva la ejecución de cada operación. De esta manera, disminuye el consumo energético del simulador a la vez que aumenta su rendimiento, lo que permite realizar simulaciones con grandes estructuras de datos. En la cache escalar, además, hay una estructura de datos adicional que representa el *write buffer*, donde se almacenan las direcciones de las líneas que han sido expulsadas a la espera de escribir sus sectores en memoria principal.

La funcionalidad principal de cada cache es la búsqueda del dato o *lookup*. Cada una posee dos versiones de esta; una para las referencias nativas de instrucciones escalares y otra para las referencias cruzadas.

4.3.1. *Lookups* nativos

El algoritmo del *lookup* nativo de la cache escalar sigue los siguientes pasos.

1. Busca la etiqueta de la dirección del dato en las líneas de su conjunto y en el *write buffer*.
2. Si coincide con una línea del conjunto y su sector es válido, se considera acierto y se continúa en el *Paso 7*.
3. Si no coincide con una línea del conjunto, pero sí con una del *write buffer*, se **restaura** la línea, se considera acierto y se continúa en el *Paso 7*.
4. Si no, se realiza la petición de *lookup* cruzado a la cache vectorial.
5. Si se encuentra en la cache vectorial, se considera acierto cruzado en la cache vectorial y se continúa en el *Paso 8*.
6. Si no, se realiza la petición a memoria principal y se considera fallo. Si la línea no está presente en la cache escalar, se **victimiza** una. Cuando el dato llega de memoria principal, se valida su sector y se continúa en el *Paso 7*.
7. Si la instrucción era una escritura, se establece el bit de modificación del sector.
8. Finaliza el *lookup*.

Para **restaurar** una línea del *write buffer* de la cache escalar se debe victimizar otra del conjunto correspondiente, en cuya posición se establecerá la línea con el sector válido referenciado que vuelve del *write buffer*.

El proceso de **victimización** en la cache escalar implica escoger como víctima la línea LRU del conjunto correspondiente para establecer en su lugar la línea del dato referenciado (su etiqueta correspondiente). Si el sector de la línea victimizada es válido y está modificado, esta debe añadirse antes al *write buffer*. En caso de que el *write buffer* esté lleno, se debe enviar una petición de *write-back* del sector de la línea más antigua de este a memoria principal. Cuando finaliza la transferencia del sector y queda un hueco libre en el *write buffer*, se puede incorporar la línea victimizada correspondiente, finalizando así la victimización.

Los pasos del algoritmo del *lookup* nativo de la cache vectorial se describen a continuación.

1. Busca la etiqueta de la dirección del dato en todas las líneas de la estructura de la cache, que ya incorpora el *write buffer*.
2. Si coincide con una línea que no es del *write buffer* y su sector correspondiente es válido, se considera acierto y se continúa en el *Paso 7*.
3. Si coincide con una línea que es del *write buffer*, que no está siendo vaciada y su sector correspondiente es válido, se **restaura** la línea, se considera acierto y se continúa en el *Paso 7*.
4. Si no, se realiza la petición de *lookup* cruzado a la cache escalar.
5. Si se encuentra en la cache escalar, se considera acierto cruzado en la cache escalar y se **migra** el sector correspondiente.
6. Si no, se realiza la petición a memoria principal y se considera fallo. Si la línea no está presente en la cache vectorial, se **victimiza**. Cuando el dato llega de memoria principal, se valida su sector y se continúa en el *Paso 7*.
7. Si la instrucción era una escritura, se establece el bit de modificación del sector.
8. Finaliza el *lookup*.

La **restauración** de una línea del *write buffer* de la cache vectorial únicamente implica dejar de considerarla *write buffer*.

Para **migrar** un sector de la cache escalar a la cache vectorial, este se invalida en la primera y se valida en la segunda. Si la línea correspondiente a dicho sector no estaba presente en la cache vectorial, la migración conlleva una victimización previa.

La **victimización** en la cache vectorial consiste en elegir la línea LRU de toda la estructura de la cache que no pertenezca al *write buffer*. Si la línea víctima no tiene ningún sector válido y modificado, se sustituye directamente por la línea del sector referenciado. En caso contrario, la línea victimizada pasa a ser parte del *write buffer* y se debe victimizar una nueva línea. La incorporación de una línea al *write buffer* debe tener en cuenta la capacidad de este. Si está lleno, se procede a vaciar su línea más antigua que no esté siendo ya vaciada. El vaciado consiste en enviar, uno a uno, cada sector válido y modificado de la línea a memoria principal para su escritura. Si todas las líneas están ya en proceso de ser vaciadas, se debe esperar a que haya alguna nueva por vaciar antes de poder incorporar la antigua víctima. Dado que la cache vectorial puede recibir varias referencias simultáneas y solo existe un bus para la transferencia de sectores a memoria, se deben evitar los solapamientos en las peticiones de vaciado. Concretamente, una operación de vaciado no puede empezar en un ciclo en el que ya se esté realizando alguna transferencia de datos hacia memoria; debe esperar a que el bus esté libre.

4.3.2. *Lookups* cruzados

El *lookup* cruzado de la cache escalar consiste en buscar un dato que ha sido solicitado por una instrucción vectorial y que no está en la cache vectorial, en la cache escalar. Tiene una funcionalidad similar de búsqueda del sector tanto en el conjunto correspondiente como en el *write buffer*. En este caso, sin embargo, en caso de acierto, se transfiere el sector a la cache vectorial. A efectos de implementación esto supone bien invalidar el sector si se encontró válido en el conjunto, o bien eliminar la fila del *write buffer* si se encontró en este. En caso de no encontrarse, se considera fallo.

Finalmente, el *lookup* cruzado de la cache vectorial presenta una funcionalidad muy simple. Si la línea se encuentra y no pertenece al *write buffer*, se considera acierto. Si la línea se encuentra, pertenece al *write buffer* y no está siendo vaciada, se restaura y se considera acierto. En caso contrario, se considera fallo. En ambos casos de acierto, si la referencia procedía de una instrucción de escritura, se establece el bit de modificación del sector.

Ambas caches realizan el seguimiento de sus propios eventos mediante diferentes contadores; *penalización acumulada, núm. de referencias nativas, núm. de fallos de referencias nativas, núm. de referencias cruzadas, núm. de fallos de referencias cruzadas, ocupación del WB, núm. de restauraciones del WB* y *núm. de sectores escritos en memoria.* Al finalizar la simulación, estos contadores permiten analizar el comportamiento del benchmark para extraer estadísticas y conclusiones.

4.3.3. Vaciado asíncrono el *write buffer*

La gestión de vaciado progresivo de *write buffers,* adicional al vaciado de emergencia provocado por la victimización, se ha contemplado en las dos caches. Siempre que la cache nativa a la instrucción del dato referenciado realiza la solicitud de

este a memoria, es decir, tras el fallo en los dos *lookups* (nativo y cruzado), se comprueba la ocupación del *write buffer*. Si esta alcanza el límite establecido para cada cache, se procede a realizar el vaciado de la línea más antigua del *write buffer* que no esté siendo ya vaciada. En la cache escalar, este límite es igual al tamaño del *write buffer*, mientras que, en la vectorial, el límite está en superar la mitad de este.

Esto se debe a que, a diferencia de lo que ocurre en la cache escalar, cuyas líneas son rápidas de vaciar, puesto que solo contienen un sector, el vaciado de una línea de cache vectorial conlleva la transferencia a memoria de entre 1 y 16 sectores. Generalmente, y dado que las líneas vectoriales están diseñadas para estar mayormente pobladas, tanto por los accesos vectoriales de datos consecutivos como por el propio *prefetching*, este número casi nunca será 1. Por ello, el vaciado de una línea del *write buffer* de la cache vectorial incurre en una penalización mayor que en la cache escalar, puesto que sus líneas probablemente tendrán siempre más sectores que vaciar. Con el objetivo de evitar largos bloqueos por el vaciado síncrono, debido a este aumento de la latencia promedio, se establece este límite reducido. Así, la mayoría de los vaciados serán asíncronos y no perjudicarán al rendimiento.

Esta funcionalidad permite vaciar los *write buffers* de forma asíncrona a medida que se van llenando, tratando de ocultar las latencias de transferencia en la propia espera para la obtención del dato referenciado de memoria. De esta manera, se intenta evitar también que las instrucciones sucesivas tengan que incurrir en la latencia del vaciado síncrono de emergencia.

4.4. Memoria principal

Para implementar la memoria principal y su controlador se ha creado la clase *DRAM* de *caveat*. En esta, se ha definido la cola de peticiones de *lookup* y *write-back* realizadas desde las caches y las operaciones realizadas sobre la memoria, siguiendo el diagrama de la *Figura 1.* Además, el controlador de memoria también implementa la operación de *prefetching* definida en la *Sección 3.3*. Esta puede verse como una operación de *lookup* solicitada por el propio controlador de memoria siempre que no queden peticiones en la cola, ni se esté realizando ninguna operación. Cuando la cache vectorial recibe el sector procedente de un *prefetching*, lo valida en su línea correspondiente. Si dicha línea ya no se encuentra en la cache, por haber sido expulsada, el *prefetching* no tiene efecto.

Al completar las operaciones de *lookup* y de *prefetching*, se realiza, de manera simultánea, la transferencia del sector por el bus y el comienzo de la siguiente operación encolada. A efectos de mejorar el rendimiento, si cuando finaliza el *prefetching* de un sector de la cache vectorial, hay encolada una petición de *lookup* a este, la petición se saca de la cola y se envía su respuesta de inmediato. El seguimiento de estos casos, en los que el *prefetching* ha contribuido de manera efectiva a reducir la latencia del procesador, se realiza con un contador que mide el *núm. de lookups desencolados por el prefetching*. Aparte de este, se han definido otros contadores para los distintos eventos; *núm. de aperturas y núm. de cierres de una fila de DRAM, núm. de accesos a la DRAM, penalización acumulada* y *núm. de bloques prefetcheados*.

4.5. Modelo de tiempos

En la *Tabla 1* se recogen las latencias asociadas tanto a cada operación realizada sobre las caches y la memoria principal, como a la transferencia de sectores en los dos buses modelados. Las latencias relativas a las operaciones sobre la memoria DRAM han sido extraídas de la *Tabla 2.1* del libro de Balasubramonian [27]. La de transferencia, sin embargo, es una latencia derivada, calculada a partir del cociente entre el tamaño del sector (64 B) y el ancho del bus (256 bits).

Operación	**Penalización (núm. ciclos)**
Lookup nativo en SC, VC y WC	1
Lookup cruzado en SC y VC	1
RAS	28
CAS	11
PRE	11
Transferencia por bus de 1 sector	2

Tabla 1. *Relación de penalizaciones establecida para cada operación implementada.*

Capítulo 5. Evaluación de resultados

En este capítulo se detalla la metodología seguida para la evaluación de la propuesta. Además, se presentan y analizan los resultados obtenidos.

5.1. Metodología

La evaluación de la propuesta de la Cache Bicameral se ha realizado comparando su rendimiento frente al de una cache escalar estándar o *white cache (WC)*, que se ha implementado como una clase de *caveat* adicional, mostrada en la *Figura 9*. La *white* cache posee las mismas características y funcionalidades que la cache escalar de la Cache Bicameral, a excepción del soporte para la petición y realización de *lookups* cruzados. En esta, se realizan todas las referencias de cualquier tipo de instrucción, y, cuando no se encuentra el dato, ni en su estructura, ni en su *write buffer*, se solicita directamente la petición a memoria principal. Para una adecuada comparación, su tamaño es el mismo que el de la Cache Bicameral; 128 KB.

Del mismo modo, la efectividad del *prefetching* se ha comparado respecto de la cota máxima que supondría una versión ideal de este. Concretamente, esta versión, que ha sido implementada también como una funcionalidad opcional del simulador únicamente con el objetivo de medir la cobertura máxima esperable para el *prefetch* realista, consiste en traer de golpe de memoria principal todos los sectores de una línea vectorial cada vez que se trae uno de esta, pero con la misma latencia. De esta manera, se simula el comportamiento idealista de un *prefetch* que consigue rellenar, automáticamente y sin aumento de la penalización, la línea vectorial entera con el primer fallo de calentamiento en esta que suponga traer un sector de memoria.

Los benchmarks empleados en la evaluación han sido las aplicaciones vectoriales mostradas en la *Tabla 2*.

Aplicación	Ámbito	Modelo algorítmico	Origen
matrizXvector	Cálculo numérico	Álgebra lineal densa	Propio
axpy	Computación de alto rendimiento (HPC)	BLAS	[23]
blackscholes	Análisis financiero	Álgebra lineal densa	[23]
jacobi-2D	Ingeniería	Álgebra lineal densa	[23]
pathfinder	Recorrido de mallas	Programación dinámica	[23]

Tabla 2. *Aplicaciones vectoriales utilizadas en la evaluación.*

Los parámetros utilizados en la evaluación de cada benchmark son los siguientes:

- *matrizXvector*. Lado de la matriz: 4096
- *axpy*. Tamaño de vector en KB: 2048
- *blackscholes*. Hilos: 1, fichero de entrada: *in_512.input*
- *jacobi-2D*. N: 32 TSTEPS: 1
- *pathfinder*. Filas: 1024, columnas: 128

Las latencias de las instrucciones de acceso a memoria se derivan del modelo de la jerarquía de memoria implementado en el simulador. Sin embargo, con el objetivo de realizar simulaciones más realistas y precisas, se han estimado también las latencias para el resto de las instrucciones vectoriales de RISC-V V que emplean dichas aplicaciones. Dicha estimación, mostrada en la *Tabla 3*, se ha realizado teniendo en cuenta la complejidad de cada tipo de operación, así como las características supuestas del pipeline de un procesador comercial con extensión vectorial como el descrito en el trabajo de Cheng *et al.* [28]. Por simplicidad, se ha considerado una penalización de 1 ciclo para el resto de las instrucciones escalares.

Instrucción	**Penalización (núm. ciclos)**
vfadd.vv	4
vfmul.vf	4
vmv1r.v	4
vslide1down.vx	8
vslide1up.vx	8
vadd.vv	4
vmin.vv	7
vmul.vv	4
vfmul.vv	6
vmv.v.i	4
vredsum.vs	8
vmv.x.s	4
vfmacc.vf	6
vadd.vx	4
vand.vx	4
vfsgnjx.vv	4
vfadd.vf	4
vfadd.vv	4
vfcvt.f.x.v	4
vfcvt.x.f.v	4
vfdiv.vv	25
vfmacc.vv	6
vfmadd.vv	6
vfmax.vf	8
vfmax.vv	8
vfmerge.vfm	4
vfmin.vf	8
vfmul.vf	4
vfmv.v.f	4
vfsgnjn.vv	4
vfsqrt.v	25
vfsub.vf	4
vfsub.vv	4
vmerge.vvm	4

vmfle.vv	4
vmflt.vv	4
vmseq.vv	4
vmv.v.x	4
vor.vv	4
vsll.vi	4
vsrl.vi	4
vsub.vx	4

Tabla 3. *Relación de penalizaciones asociadas a cada instrucción vectorial de los benchmarks evaluados.*

Además de variar el tipo de benchmark, se ha modificado también el tamaño máximo de vector soportado por la arquitectura. Esto permite evaluar el rendimiento de la propuesta en procesadores con diferente tamaño de registros vectoriales. Específicamente, se han empleado longitudes de vector de 256, 512, 1024 y 2048 bits.

Para ejecutar los diferentes tipos de simulación, se han especificado por línea de comandos las distintas opciones mostrados a continuación:

```
caveat --cores=1 --vec=vlen:<bits_vlen>,elen:64,slen:<bits_vlen>
       --bicameral=<bc> --prefetch=<tipo_prefetch> <app> <args>
```

<bits_vlen>
: Determina el número de bits de la longitud de vector arquitectural.

<bc>
: Si *true*, habilita la simulación con la Cache Bicameral. Si *false*, habilita la simulación con la white cache. Por defecto, se establece a *true*.

<tipo_prefetch>
: Determina el tipo de *prefetching* empleado en la simulación con Cache Bicameral. Los valores de opción posibles son:
  - *0*: sin *prefetching*. Opción establecida por defecto.
  - *1*: con *prefetching* ideal (todos los sectores de la línea de una vez)
  - *2*: con *prefetching* realista (del controlador de memoria).

<app> <args>
: Determina la aplicación a simular y sus argumentos.

5.2. Resultados

En esta sección se recogen los valores de *speedup*, tiempo medio de acceso a memoria, aperturas de filas de DRAM y relación de aciertos y fallos obtenidos tras cada simulación.

5.2.1. Speedup

En la *Figura 10* se muestra el *speedup* de la Cache Bicameral frente a la white cache, tanto sin el empleo de *prefetching* como con él, en cada benchmark evaluado con cada longitud de vector, además del promedio. La barra horizontal determina la cota

superior de *speedup* posible que se alcanzaría en caso de *prefetching* ideal, en el cual, en vez de un único sector, se traerían, con la misma latencia, todos los sectores de la fila vectorial a la vez.

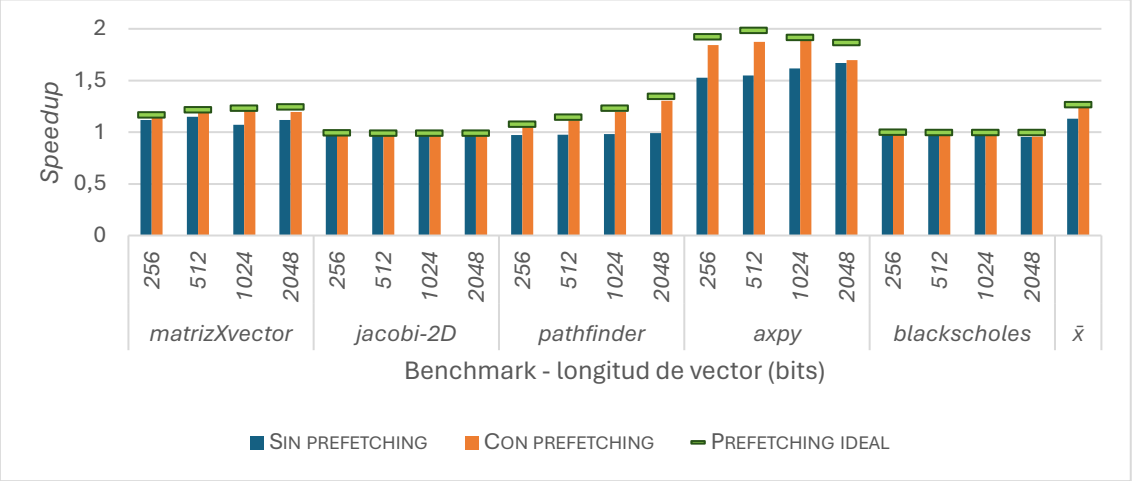

Figura 10. *Speedup de cada benchmark con cada longitud de vector.*

La principal observación que se puede extraer de estos resultados es que la Cache Bicameral mejora de media en un 1,24 respecto a la white cache en el caso de utilizar el *prefetching* y en un 1,13 sin él. En tres de los cinco benchmarks evaluados la cache propuesta proporciona mejores resultados que la convencional, mientras que, en los otros dos, iguala estos resultados, salvo en el caso de *blackscholes* con un tamaño de vector de 2048, en el que el *speedup* es ligeramente inferior a uno (0,96). Adicionalmente, los resultados obtenidos con el *prefetching* están muy cercanos a su cota ideal, lo que significa que la Cache Bicameral consigue ocultar con éxito su latencia. En conjunto, estos resultados suponen una mejora importante, ya que se consiguen sin necesidad de añadir hardware adicional ni incrementar el tamaño de cache, simplemente de una reestructuración y un mejor aprovechamiento de los recursos disponibles.

Otra conclusión general es que el incremento del tamaño del vector parece tener un impacto positivo (o bien no tiene impacto, como en *jacobi-2D* y *blackscholes*), salvo para el tamaño de 2048 bits, que produce un ligero descenso del *speedup* (a excepción del caso de *pathfinder*) respecto a tamaños menores. Esto último se puede deber al incremento del tiempo medio de acceso a memoria que se muestra en la siguiente sección.

Analizando los benchmarks por separado, tanto en *matrizXvector* como en *axpy*, se consigue siempre mejor rendimiento con la propuesta de Cache Bicameral, independientemente de la longitud de vector. Para *axpy*, supone un incremento en el rendimiento, en todos los casos, superior al 50%.

Por su parte, en *pathfinder*, la Cache Bicameral únicamente logra mejorar el rendimiento con el empleo de *prefetching*, y este se ve influenciado, de manera muy positiva, con el aumento de la longitud de vector arquitectural.

En el resto de casos, la modificación del tamaño de vector no resulta significativa, a excepción del tamaño máximo en *axpy*, que parece reducir la eficiencia del *prefetching*. A la vista de la gráfica, se observa que el *prefetching*, además de no resultar perjudicial en ninguna ocasión, en la mayoría de ocasiones, aumenta el *speedup* hasta valores muy cercanos, incluso idénticos, a la cota ideal.

Finalmente, en *jacobi-2D* y *blackscholes*, la propuesta de la Cache Bicameral no parece aportar valor adicional, probablemente porque los tamaños de problema elegidos han sido demasiado pequeños.

5.2.2. Tiempo medio de acceso a memoria

En la *Figura 11* se muestra el tiempo medio de acceso a memoria, representado en número de ciclos en el rango de 0 a 10 del eje Y. Para cada combinación de benchmark con longitud de vector, además de para la media, en el eje X, se muestran 3 valores de tiempo medio; los de la Cache Bicameral con y sin *prefetching* y el de la white cache.

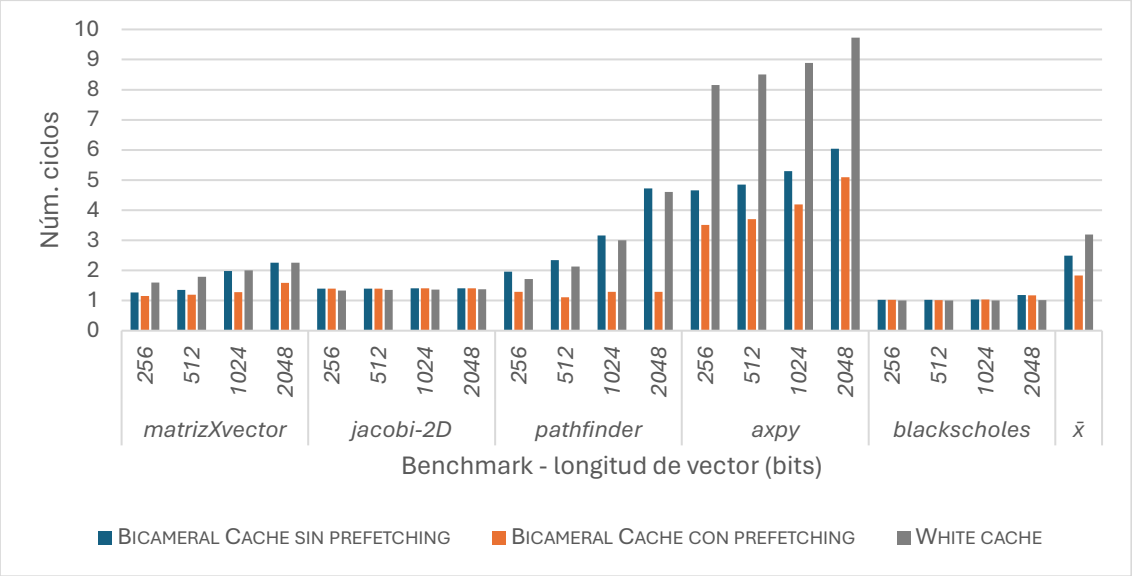

Figura 11. *Tiempo medio de acceso a memoria de cada benchmark con cada longitud de vector.*

De la gráfica se extrae la conclusión de que la Cache Bicameral permite reducir la latencia media de acceso a memoria de manera general en de 3,19 ciclos a 2,49 sin *prefetching* y 1,83 con este, lo que justifica las mejoras en rendimiento vistas en el apartado anterior.

De esta gráfica, destaca el caso de *axpy*, en el que, mientras que con la cache convencional, el tiempo medio de acceso no baja de 8 ciclos, con la Cache Bicameral se llega a un máximo de 6. Los resultados en este caso van acordes a los vistos previamente con el *speedup*; *axpy* es el benchmark en el que mayor impacto positivo tiene la Cache Bicameral. Del mismo modo, el *prefetching* ayuda a reducir siempre su tiempo medio de acceso a memoria, lo que justifica las consecuentes mejoras en el *speedup* de la gráfica anterior. El incremento del tiempo de acceso a memoria con el tamaño del vector, sin embargo, no tiene un impacto importante sobre el speedup, salvo en el caso de vectores muy grandes (2048 bits), como se puede apreciar en los resultados mostrados en la sección anterior (*Figura 10*).

Para *matrizXvector*, con longitudes de vector de 1024 y 2048, no se aprecia reducción en el tiempo medio de acceso, a pesar de las ganancias en el rendimiento vistas anteriormente, en ausencia de *prefetch*. En los otros dos casos, la Cache Bicameral por sí sola ya logra reducir la latencia de acceso a memoria, si bien, de manera consistente, el *prefetching* proporciona mejores resultados.

En el caso de *pathfinder*, la Cache Bicameral sin *prefetch* incrementa ligeramente el tiempo medio de acceso a memoria, provocando esa pérdida de rendimiento respecto de la white cache comentada anteriormente. Al introducir dicha optimización, esta latencia se estabiliza, acercándose a 1. Es aquí donde, al incrementar la longitud de vector, se nota cada vez más ganancia, al igual que pasaba con el *speedup*, frente al modelo convencional.

Sobre lo observado en *jacobi-2D* y *blackscholes*, se extrae la conclusión de que el tiempo medio de acceso a memoria es ya de por sí lo suficientemente bajo como para que resulten indiferentes no solo la Cache Bicameral sino también su *prefetching*. Concretamente, en *blackscholes*¸ la latencia media de memoria es 1, lo que indica que, efectivamente, el *working set* utilizado en este experimento es muy pequeño y cabe completamente en la memoria cache (menor de 128 KB), por lo que la tasa de fallos es muy pequeña y, por lo tanto, cualquier modificación que se haga en la jerarquía de memoria, no va a tener impacto en el rendimiento. Esto puede explicar también que apenas se aprecien diferencias de rendimiento al modificar la longitud de vector.

5.2.3. Número de aperturas de filas de DRAM

En esta sección se presentan los resultados obtenidos respecto al número de aperturas de filas DRAM de los benchmarks *matrizXvector* y *axpy (Figura 12), pathfinder (Figura 13), y jacobi-2D y blackscholes (Figura 14),*para cada longitud de vector evaluada. Ha sido necesario dividir los resultados en tres gráficos debido a las grandes diferencias en los rangos de sus valores, que provocan que estos no se puedan apreciar bien usando una sola gráfica con un único eje Y. En el eje horizontal, se representa una columna por cada configuración; Cache Bicameral con y sin *prefetching* y white cache.

Esta métrica es interesante de analizar ya que ayuda a entender mejor el comportamiento de la memoria principal con cada benchmark. En el modelo de DRAM empleado, cada apertura de una de sus filas conlleva el cierre de la que se encontraba abierta anteriormente. Si estas dos operaciones se producen habitualmente con el core bloqueado, es decir, en mitad de una petición de memoria, la latencia promedio aumenta, empeorando por tanto el rendimiento, dado que para obtener el dato no se estaría leyendo directamente (CAS), sino que se estaría cerrando una fila, abriendo otra y leyendo (PRE+RAS+CAS). Reducir el número de filas abiertas, a consecuencia, suele ser beneficioso para el rendimiento general.

La gráfica de la *Figura 12* muestra valores del orden de los millones. Para *matrizXvector*, el número de aperturas de la Cache Bicameral es, en la mayoría de casos, muy interior al medio millón, por lo que apenas se aprecian sus columnas. En ella, ambos benchmarks revelan una muy notable diferencia entre el número de aperturas de filas de la DRAM que realiza la cache convencional frente a las que realiza la Cache Bicameral. En el caso de *matrizXVector*, el número de la primera es, al menos, 8 veces superior; siendo del orden de los 4 millones en todos los casos. Para *axpy*, si bien los valores son más cercanos entre sí, la white cache sigue requiriendo aproximadamente tres veces más que la Bicameral.

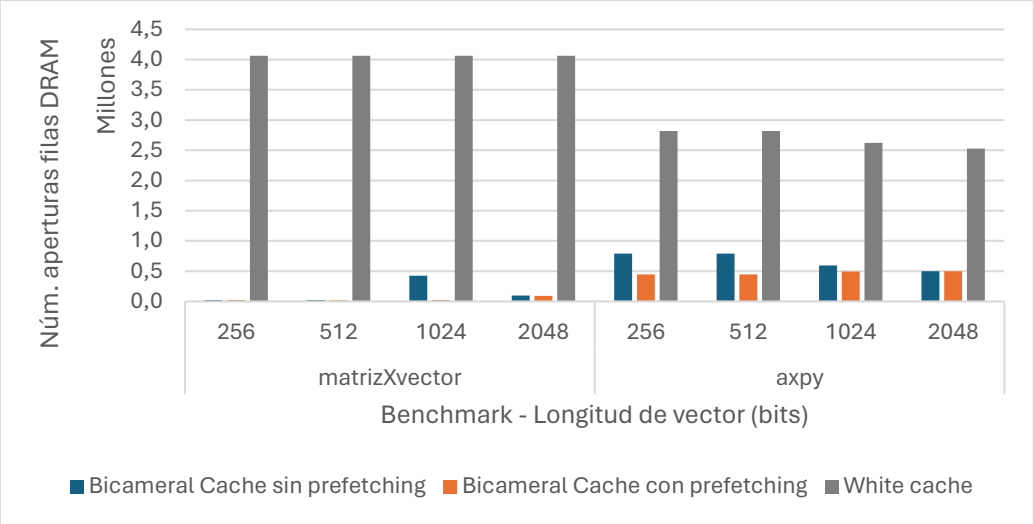

***Figura 12**. Número de aperturas de filas de DRAM en matrizXvector y axpy.*

Para *matrizXvector*, sin embargo, esta enorme disparidad no se traduce en una equivalente mejora de rendimiento. De hecho, como se apreció en la ilustración anterior, ni siquiera conlleva un aumento excesivo de la diferencia en el tiempo medio de acceso a memoria. Una posible explicación a este fenómeno sería que, mientras en la cache estándar algunos datos se están reemplazando continuamente, debido a colisiones entre los distintitos tipos de instrucción de memoria —escalar y vectorial—, en la Bicameral se mantienen segregadas, y, por tanto, libres de interferencias entre sí. A pesar de ello, el número de veces que estos datos son reemplazados, es decir, el número de interferencias entre los dos tipos de instrucción es lo suficientemente pequeño en el conjunto total de referencias como para que, si bien el número de aperturas de filas parezca inmenso, el promedio en la latencia de acceso no se dispare.

En *axpy*, por el contrario, la diferencia en el número de aperturas de filas de la memoria principal sí parece estar correlacionada con la latencia media de acceso y, consecuentemente, con el rendimiento. De nuevo, esto puede significar que la interacción entre las instrucciones escalares y vectoriales en una misma cache tiene un efecto negativo. En este caso, se deduce que los datos de un tipo de referencias que son reemplazados por la interferencia del otro tipo provocan que el tiempo medio de acceso a memoria, y a causa de ello, el rendimiento, se vean perjudicados. Esto puede ser causado, bien porque los datos reemplazados tengan que volver a ser usados, lo que aumentaría la tasa de fallo, o bien porque el reemplazo conlleve post escrituras síncronas en memoria de los datos de *write buffer*, debido a los reemplazos de la victimización.

Por último, cabe destacar que la variación en el tamaño de los registros vectoriales del procesador no parece influir en exceso en el número de aperturas de filas de DRAM necesarias. Únicamente en *axpy* se puede apreciar una ligera disminución de dicha cantidad a medida que aumenta la longitud vectorial. En el *matrizXvector*, si bien se aprecia un incremento de aperturas con las dos longitudes de vector más largas, especialmente para 1024 bits con la Cache Bicameral sin *prefetching*, no parece existir un patrón claro de relación.

A continuación, se muestra, en la *Figura 13*, la gráfica que recoge la misma métrica en el caso del benchmark *pathfinder*.

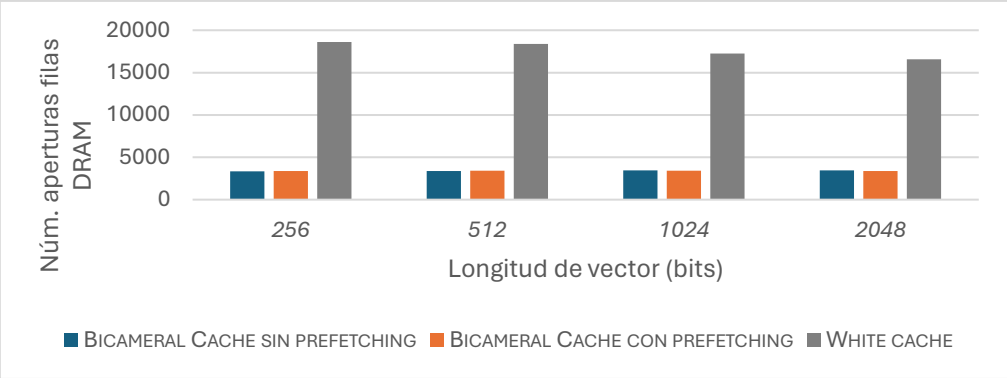

Figura 13. *Número de aperturas de filas de DRAM en pathfinder.*

A pesar de que, al analizar las dos gráficas anteriores para este benchmark, la Cache Bicameral no lograba obtener mejora, ni en *speedup*, ni en tiempo medio de acceso a memoria, respecto de la white cache en ausencia de *prefetching*, en la métrica reflejada en esta gráfica no se observa ese hecho. En este caso, en las dos versiones de Cache Bicameral se necesitan, aproximadamente, 4 veces menos de aperturas. De ello, se puede deducir que, mientras estas dos versiones conllevan el mismo número de aperturas de filas de DRAM, la versión sin *prefetching* tiene una cantidad inferior de aciertos nativos, es decir, de primer nivel, que justifica el aumento en su latencia media de acceso visto anteriormente. Además, los sectores sobre los que se realiza el *prefetching* pertenecen siempre a filas ya abiertas, lo que supone que, al incluir este mecanismo, se consigue mayor rendimiento, como se apreciaba en la gráfica de *speedup*, porque se traen, por adelantado, los sectores que se van a necesitar sin aumentar el número de aperturas.

Sobre el efecto de la longitud de vector, se puede determinar que las longitudes largas parecen favorecer ligeramente la reducción de aperturas en la white cache, si bien dicho efecto es insuficiente para disminuir su orden de magnitud. En la Cache Bicameral, no se aprecia diferencia.

Finalmente, en la *Figura 14* se representa, de la misma forma, el número de aperturas de filas de DRAM para el resto de benchmarks; *jacobi-2D* y *blackscholes*.

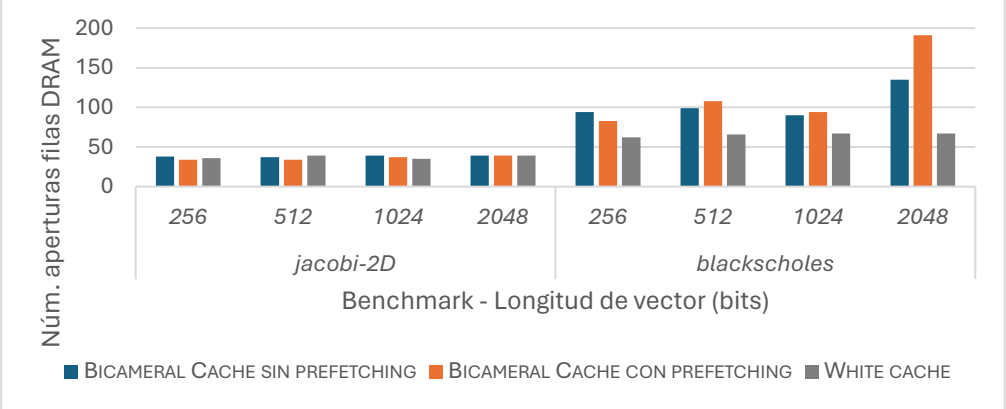

Figura 14. *Número de aperturas de filas de DRAM en jacobi-2D y blackscholes.*

En *jacobi-2D* se observa la misma tendencia que aparecía al analizar las dos métricas anteriores. Los tres tipos de versiones de cache evaluados suponen prácticamente

el mismo número de aperturas de filas de DRAM, independientemente del aumento en el tamaño de vector. Este efecto deriva en el similar tiempo medio de acceso a memoria que se reflejaba en la *Figura 11*, que a su vez igualaba el rendimiento obtenido, no solo en los tres casos mencionados, sino también para la cota máxima ideal del *prefetching*. La razón de estos valores en dicho benchmark se debe a dos factores; una reducida tasa de fallos que justifica tanto la escasez de aperturas de filas de DRAM como la baja latencia promedio de acceso a memoria, y una muy reducida interferencia entre las instrucciones escalares y vectoriales que se traduce en un rendimiento constante en todos los casos. Asimismo, teniendo en cuenta que el *prefetching* ideal tampoco supone ninguna mejora a este rendimiento, se puede deducir que la presencia de instrucciones vectoriales de acceso a memoria en la aplicación es ínfima; ni hay muchas referencias vectoriales a datos consecutivos que se aprovechen del *prefetch*, ni muchas con *stride* que aumenten la latencia promedio.

En cuanto al número de aperturas de *blackscholes*, se puede observar que, para la white cache, se mantiene siempre constante indistintamente de la longitud de vector. Al incluir la Cache Bicameral, sin embargo, este número aumenta, y aún más al añadir el *prefetch* para todos los tamaños de vector excepto el menor. Esta diferencia se agranda con el máximo tamaño de vector, provocando el ligero aumento en la latencia promedio de acceso percibido en el análisis de la *Figura 11* y su consecuente pérdida de rendimiento mostrado en la *Figura 10*. Para el resto de casos, este aumento de aperturas no perjudica la latencia de acceso, que de media equivale a 1 solo ciclo, es decir, aciertos siempre en el primer nivel de la jerarquía.

El motivo del aumento en el número de aperturas de filas de DRAM que se realizan al emplear la Cache Bicameral no se debe a un aumento de la tasa fallos respecto de la cache convencional, dado que se sigue manteniendo, para las tres primeras longitudes, la misma mínima latencia posible; 1 ciclo. Dicho aumento está motivado por el vaciado asíncrono de bloques de *write buffer*, que provoca nuevas aperturas de filas en memoria, pero no perjudica el tiempo de acceso en las lecturas y escrituras. Al tener cada una de las caches que forman la Cache Bicameral la mitad de tamaño que la white cache, sus respectivos *write buffers* se llenan antes debido al aumento de los reemplazos, lo que provoca, simplemente, más peticiones de post escritura a memoria, con sus consecuentes aperturas de fila.

Además, al igual que en el benchmark anterior, se puede deducir también una muy escasa presencia de operaciones vectoriales de acceso a memoria en la aplicación, que justifique la ausencia de mejora con la inclusión de *prefetching*, a la vez que la misma latencia de acceso promedio y el mismo rendimiento que con la cache convencional. De ello se extrae que el *write buffer* que está provocando tales vaciados en la Cache Bicameral es el de su partición escalar, dado que, a falta de un gran número de referencias vectoriales en *blackscholes*, es la que más reemplazos está sufriendo con respecto de la white cache, únicamente motivados por su tamaño y por la notable mayoría de instrucciones escalares de acceso a memoria.

5.2.4. Relación de aciertos y fallos

En este apartado se analiza la relación entre fallos y aciertos en cada caso evaluado por cada uno de los benchmarks.

El primer desglose a analizar es el de *matrizXvector*, recogido en la *Figura 15*. La gráfica muestra, en el eje Y, el porcentaje de fallos y aciertos de cada tipo que resultan del total de referencias en los tres casos analizados; las dos versiones de Cache Bicameral —sin *preftching* (BC SIN) y con *prefetching* (BC PF)— y la white cache (WC), para las cuatro longitudes de vector contempladas. En cada columna se identifica el resultado de cada posible referencia en la cache con un color; azul para aciertos de referencias escalares, naranja para aciertos cruzados en la cache escalar, verde para aciertos de referencias vectoriales, amarillo para a ciertos cruzados vectoriales y rojo para fallos.

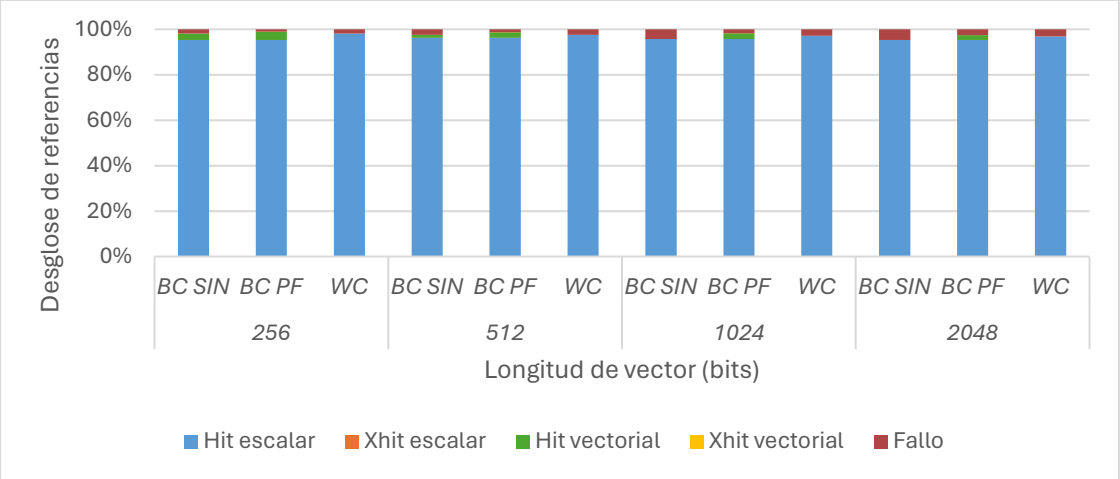

***Figura 15**. Desglose de referencias en matrizXvector.*

A la vista de la gráfica, destaca la ausencia de aciertos cruzados en la Cache Bicameral. Resulta también interesante la semejanza entre las tasas de fallos entre la cache propuesta y la convencional. En este caso, si bien el número fallos parece ser el mismo, la latencia media de acceso a memoria y, especialmente, el número de aperturas de filas de DRAM en la white cache, como se vio con anterioridad, son mucho más elevadas.

Por ello, la causa de la mejora de rendimiento que se alcanza con la Cache Bicameral parece estar motivada por las interferencias entre las instrucciones vectoriales y escalares. Que no haya presencia de referencias cruzadas demuestra que los datos de los vectores no son utilizados por las referencias escalares, y viceversa. Eso significa que los dos tipos de instrucción tienen bien acotados el conjunto de los datos que utilizan. Sin embargo, puede deducirse que, en la white cache, estos datos producen colisiones en los conjuntos. Esto explicaría que se produjeran reemplazos modificados de datos que, aunque no fueran a volver a ser usados, como se intuye de la inalterabilidad de la tasa de fallo, provocaran un aumento de la latencia de acceso debido al vaciado de *write buffer*. La causa de dicho aumento es que esos vaciados son síncronos, es decir, están provocados por la victimización y, además, deben ser escritos en otra filas de memoria distintas a la abierta en cada momento, de ahí los valores observados en las gráficas de la *Figura 11* y la *Figura 12*. En la Cache Bicameral, esta interferencia no existe pues, al estar segregada, un tipo de referencia nunca pueden reemplazar los datos del otro tipo.

Se observa también el comportamiento esperado del *prefetching* tras el análisis de la gráfica de *speedup* inicial; este reduce la tasa de fallo la aumentar el número de aciertos vectoriales en la Cache Bicameral.

El mismo desglose se muestra para las evaluaciones de *jacobi-2D* en la *Figura 16*.

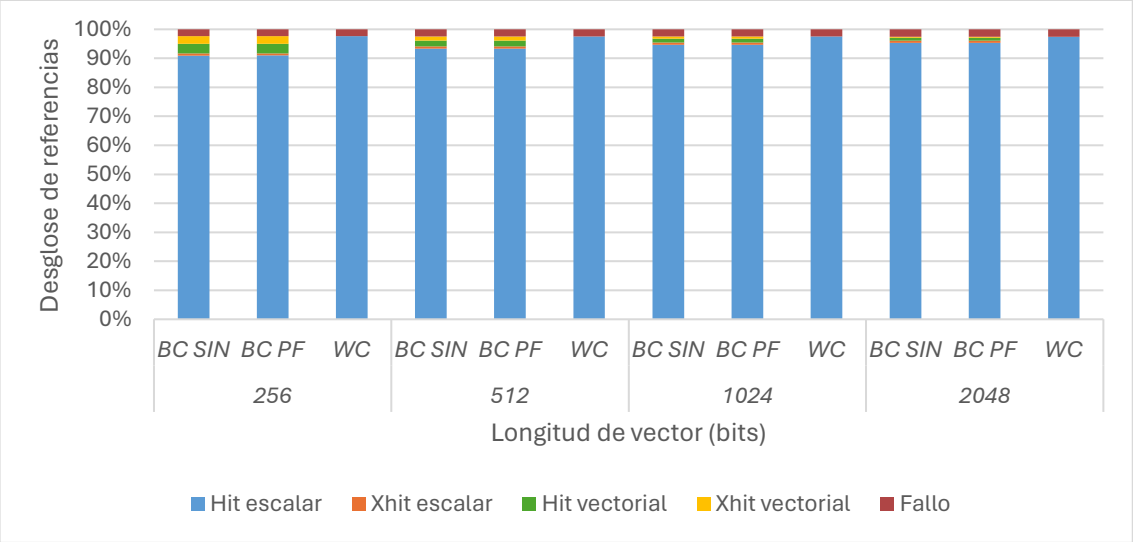

Figura 16. *Desglose de referencias en jacobi-2D.*

En este caso, a diferencia de lo que ocurría en *matrizXvector*, sí aparecen, para las cuatro longitudes de vector, aciertos cruzados. Sin embargo, como ya se sabía tras el análisis de su *speedup*, este benchmark no se aprovecha en absoluto del *prefetching*; la cantidad de aciertos vectoriales en las dos versiones de la Cache Bicameral es constante, probablemente debido al patrón de acceso de las respectivas referencias. Además, y de nuevo, confirmando lo visto anteriormente, la cantidad de fallos es constante, razón por la cual la cache propuesta no aporta ventaja sobre el rendimiento.

Para concluir el análisis, cabe destacar que, en línea con lo que ya se comentó, el comportamiento de la simulación a medida que se incrementa la longitud de vector no se ve especialmente afectado, si bien sí se aprecia, en la Cache Bicameral, una disminución progresiva en el porcentaje de aciertos vectoriales y aciertos cruzados escalares. La motivación parece estar clara; si bien el número de fallos por referencias vectoriales no varía, la cantidad de estas sí. Al trabajar con vectores más largos, se necesitan menos referencias para manejarlos y, por tanto, disminuye la relación de aciertos. Este comportamiento se ejemplifica claramente con el diagrama de la *Figura 17*; cuanto mayor es la longitud de vector, menos referencias se necesitan, pero cada una es de mayor tamaño; por lo que, si bien produce los mismos fallos, consigue menos aciertos en total.

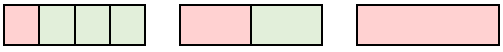

Figura 17. *Representación de la disminución de aciertos al aumentar el tamaño de la referencia. Cada sección simboliza una referencia y, en total, se accede al mismo tamaño de datos. De izquierda a derecha; 4 referencias (1 fallo y 3 aciertos), 2 referencias (1 fallo y 1 acierto) y 1 referencia (1 fallo).*

A continuación, se muestra la gráfica de la *Figura 18* con la relación de aciertos y fallos en las simulaciones de *pathfinder*.

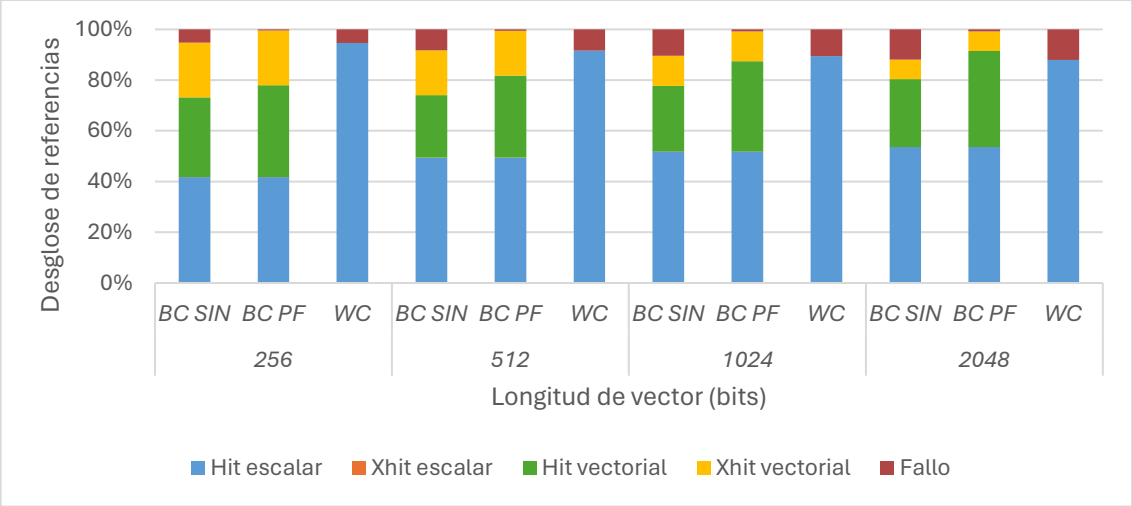

Figura 18. *Desglose de referencias en pathfinder.*

En comparación a las dos gráficas anteriores para esta misma métrica, en la de *pathfinder* destaca la enorme presencia de aciertos vectoriales, tanto nativos como cruzados. Las franjas de estos dos colores, verde y amarillo, respectivamente, eran mucho menos relevantes en los diagramas previos. Este incremento parece indicar que la aplicación tiene mucho reúso de sus datos vectoriales, que vuelven a ser referenciados por instrucciones no solo de tipo vectorial, sino también de tipo escalar, a juzgar por dichas franjas.

Como ya se anticipó en el análisis de su cantidad correspondiente de filas abiertas de DRAM, la Cache Bicameral sin *prefetching* tiene una menor cantidad de aciertos nativos que la white cache, a pesar de que su número de fallos sea idéntico. Esto significa que, si bien la relación total de aciertos en cache es parecida, en la Cache Bicameral hay muchos menos aciertos de primer nivel que suponen una penalización de 1 único ciclo (franjas azules y verdes). Por su parte, todos los aciertos vectoriales cruzados (franja amarilla), conllevan una penalización de 2 ciclos, puesto que han sido implementados de manera secuencial en el simulador. De esta forma, estos aciertos conllevan 1 ciclo de *lookup* en la cache escalar que resulta en fallo y 1 ciclo de *lookup* cruzado en la cache vectorial que encuentra el dato. A pesar de ello, se podría haber optado por una implementación, totalmente factible en hardware, de ambos *lookups* en paralelo. Esta alternativa proporcionaría mayor rendimiento de búsqueda en la cache a base de incrementar el consumo energético. Sin embargo, para el caso de *pathfinder*, a la vista de la gráfica se puede determinar que no aportaría ventaja adicional para la Cache Bicameral, dado que, en ausencia de *prefetching*, consigue la misma cantidad de fallos que la white cache. Es únicamente al introducir el *prefetch* cuando se consigue reducir casi por completo la presencia de fallos, a excepción de aquellos ineludibles por calentamiento, es decir, por el primer acceso vectorial al sector de cada fila vectorial referenciada y por el primer acceso escalar que no varía.

Con el aumento de la longitud vectorial, parece apreciarse un aumento de la tasa de fallos en la versión *sin prefetching* y en la white cache. La explicación, al igual que en caso de *jacobi-2D*, se debe a la reducción en el número total de referencias vectoriales causado por el incremento de longitud del vector arquitectural. De este modo, si bien los fallos son realmente los mismos, al producirse menos referencias y, por tanto, menos

aciertos, la tasa de fallos se ve incrementada, pues ha aumentado la relación de número de fallos respecto del total de referencias. En la versión de Cache Bicameral con *prefetching*, este incremento en la tasa de fallos no es tan apreciable debido al propio efecto positivo del *prefetch* que, al incrementar el número de aciertos totales, ayuda a compensar la diferencia entre el número de fallos y de aciertos.

El siguiente desglose de referencias, recogido en la *Figura 19*, corresponde al benchmark *axpy*.

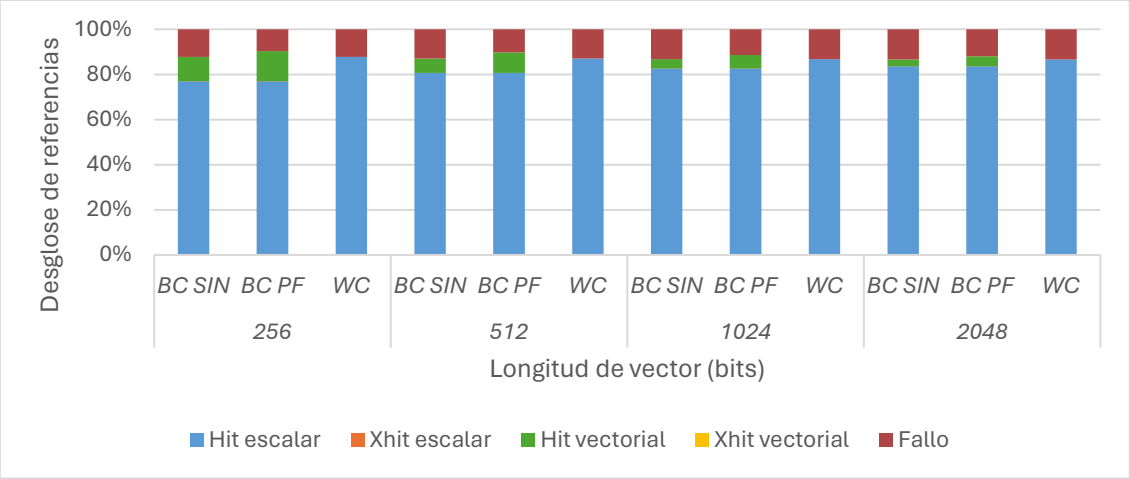

***Figura 19**. Desglose de referencias en axpy.*

En *axpy*, ocurre parecido a lo visto en *matrizXvector*, a pesar de observarse la misma tasa de fallos en la Cache Bicameral sin *prefetching* y en la white cache, la segunda tiene un peor rendimiento motivado por una mayor latencia promedio de acceso a memoria. De igual modo, el número de aperturas de filas de DRAM resultó ser también superior en la cache convencional. Por su parte, las referencias cruzadas, no constituyen un volumen relevante en el desglose de referencias.

Por todo esto, se deducen las mismas conclusiones que para *matrizXvector*; la mejora de rendimiento de la Cache Bicameral frente de la white cache, en ausencia de *prefetch*, se debe a la segregación de las instrucciones escalares y vectoriales. De esta forma, entre ellas no se expulsan los datos mutuamente, algo que sí parece ocurrir en la cache convencional, provocando grandes latencias a memoria por realizar los vaciados de *write buffer* de manera síncrona, es decir, bloqueante. El mismo razonamiento aplica sobre las filas de DRAM a las que pertenecen estos sectores vaciados; estas no coinciden con las abiertas en cada caso, por lo que se debe realizar una operación de cerrado (PRE) y otra de apertura (RAS). Estas últimas son las que suponen la gran diferencia entre ambas caches evidenciada en la gráfica de la *Figura 12*. En base al tiempo medio de acceso a memoria, el número de interferencias es, en *axpy*, mucho mayor que en *matrizXvector*, lo que motiva que la Cache Bicameral, en este benchmark, consiga ganancias en *speedup* mucho mayores frente a la cache convencional.

En cuanto a la longitud de vector, su impacto al aumentarla en este benchmark es el mismo que en los anteriores; reduce el número de referencias vectoriales, aumentando así la tasa de fallos. En la gráfica, esto se refleja como una menor relación de aciertos vectoriales, determinada por la reducción de la franja verde.

Para terminar, en la *Figura 20* se representa el desglose de cada simulación con la aplicación *blackscholes*.

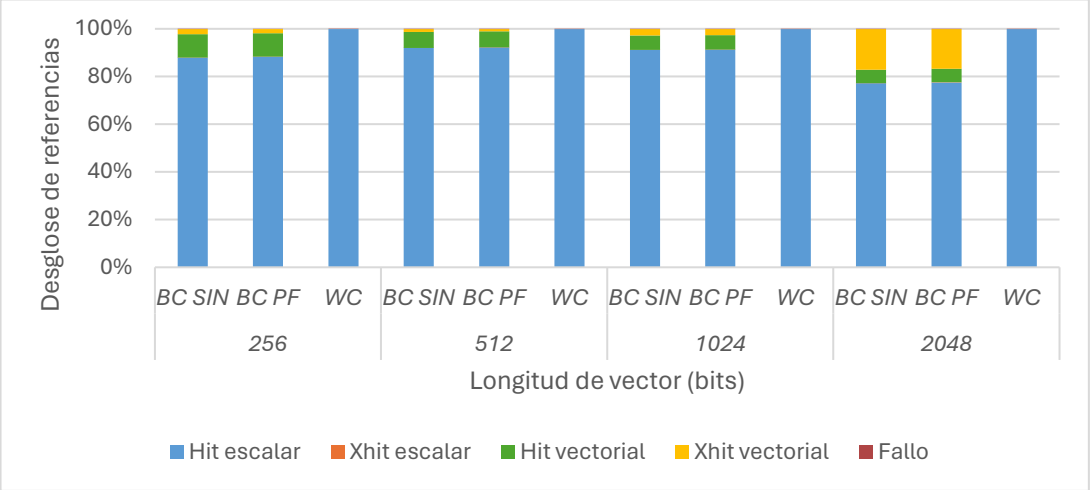

Figura 20. *Desglose de referencias en blackscholes.*

A diferencia de los desgloses del resto de benchamrks, en este destaca enormemente la ausencia de fallos. En ninguna de las tres versiones de cache evaluadas el porcentaje de fallos es lo suficientemente relevante para que su franja roja correspondiente aparezca en la gráfica. A causa de ello, en la white cache prácticamente la totalidad de las referencias sean aciertos, lo cual se traduce en la latencia promedio de 1 ciclo que se reflejaba en la *Figura 11*. En estas circunstancias, la Cache Bicameral no puede aportar ninguna mejora, pues el benchmark ya consigue aproximadamente un 100% de tasa de acierto una cache convencional. En las columnas pertenecientes a la cache propuesta, se observa que hay un porcentaje, determinado por la franja amarilla, de aciertos causados por *lookups* cruzados sobre la cache vectorial, que además aumenta con la longitud de vector. Estos aciertos, tal y como ha sido definida la implementación, conllevan el doble de latencia que los demás, pues se ha optado por modelar los *lookups* secuenciales, en vez de en paralelo, como se explicó con anterioridad.

Tras el análisis del desglose de referencias de los cinco benchmarks, cabe destacar que se producen muy pocos aciertos cruzados en al cache escalar, representados por la franja naranja, los cuales conllevan a la migración del sector referenciado hacia la cache vectorial. Esto permite concluir que el enfoque de esta migración unilateral es acertado; priorizar la presencia de los datos vectoriales en la cache vectorial evita comprometer el rendimiento de las referencias vectoriales, aunque estos sean usados también por las instrucciones escalares.

Capítulo 6. Conclusiones y trabajos futuros

En esta última sección se realiza un repaso, a modo de resumen, de la propuesta, el trabajo realizado y el análisis extraído de su evaluación. De igual manera, se ofrece una reflexión sobre las posibles líneas de investigación para continuar y ampliar este trabajo.

6.1. Conclusiones

El desarrollo de este trabajo se ha centrado en el diseño, modelado, implementación, depuración y evaluación de una nueva propuesta de cache para procesadores vectoriales.

Esta cache está compuesta de dos particiones; una cache escalar y una cache vectorial, cada una de las cuales está dedicada a las referencias de memoria de las instrucciones de su tipo correspondiente. La cache escalar es una cache estándar asociativa por conjuntos, mientras que la vectorial es totalmente asociativa. Las líneas de esta última son sectorizadas, de forma que pueden almacenar múltiples sectores, cada uno con un tamaño igual a la línea de la cache escalar. Así, la cache vectorial puede alojar, en una única línea, varios elementos de un mismo vector. Ambas caches son mutuamente excluyentes, por lo que, tras un fallo en la cache propia al tipo de instrucción de memoria que referencia el dato, se debe realizar una búsqueda en la otra cache antes de acudir a memoria principal. El orden secuencial de esta segunda búsqueda está meramente motivado por la eficiencia energética, si bien una implementación paralela es totalmente factible en hardware. Para preservar la continuidad de los vectores en las líneas de la cache vectorial, los aciertos escalares de datos referenciados por instrucciones vectoriales conllevan la migración de estos a su cache original. Con el objetivo de reducir las latencias de acceso a memoria durante las operaciones de escritura, cada cache está provista de un *write buffer* con capacidad para almacenar varias líneas expulsadas de la cache con datos válidos y modificados a la espera de ser escritos de nuevo en memoria principal. La cache vectorial posee un mecanismo optativo de rellenado progresivo de sus largas líneas vectoriales (*prefetch*), que desencadena el controlador de memoria cuando no tiene peticiones pendientes. La finalidad de esta funcionalidad adicional es abastecer a las líneas recientemente accedidas con nuevos datos para aprovechar la localidad espacial.

Para implementar la propuesta se ha empleado el simulador Cavatools de la ISA RISC-V con soporte para la extensión vectorial. La evaluación de su rendimiento se ha efectuado mediante la simulación de 5 benchmarks vectorizados distintos: *matrizXvector, jacobi-2D, pathfinder, axpy* y *blackscholes*. Con cada uno, se ha analizado el efecto de variar la longitud máxima de vector de la arquitectura del procesador simulado, para 256, 512, 1024 y 2048 bits, así como de la propuesta de Cache Bicameral, con y sin optimización de *prefetching*, frente a una cache escalar convencional del mismo tamaño.

Los análisis de las diferentes simulaciones llevadas a cabo permiten determinar que la cache propuesta ofrece mejoras de rendimiento en aplicaciones vectorizadas cuyas instrucciones vectoriales tengan el suficiente peso en la ejecución y el tamaño del problema sea lo suficientemente grande para poder aprovechar sus características, como

son *matrizXvector* y, en especial, *axpy*. Por el contrario, la simulación de *blackscholes* es un claro ejemplo en el que, si bien se efectúan numerosas referencias vectoriales, el conjunto de datos empleados es tan reducido que cabe entero en la cache, sin producir fallos de colisiones, imposibilitando la explotación de los beneficios de la nueva propuesta de cache. Por otra parte, como se deduce del estudio de *jacobi-2D* y *pathfinder*, la Cache Bicameral permite erradicar las interferencias que las instrucciones escalares provocan sobre los datos vectoriales al compartir una misma cache, causantes de reemplazos no deseables que originen un aumento de las latencias de acceso a memoria. En cuanto al *prefetching*, no solo no se ha contemplado ninguna situación en la que su efecto resulte negativo, sino que, por lo general, su presencia resulta muy beneficiosa para el rendimiento, por lo que se puede considerar una optimización recomendable. La longitud de vector, por su parte, no parece jugar un papel demasiado relevante a la hora de evaluar el rendimiento de las caches; únicamente se ha observado relación en *pathfinder* con el uso conjunto del *prefetching*.

6.2. Trabajos futuros

El desarrollo de este trabajo da pie a nuevas líneas de investigación que permitan profundizar en diferentes aspectos relacionados con la Cache Bicameral, como las mencionadas a continuación.

- Implementación y evaluación de diferentes algoritmos de *prefetching*. En este trabajo, únicamente se ha explorado la versión básica; elegir el siguiente sector inválido por la derecha, si lo hubiera, del referenciado por la última petición de memoria de una instrucción vectorial. Sin embargo, existen infinidad de algoritmos de *prefetching* con mayor grado de complejidad que puedan resultar, quizá, más adecuados para algunos tipos de problema.

- Extensión de la funcionalidad del controlador de memoria para permitir tantas filas abiertas como bancos de DRAM haya disponibles. Este comportamiento es más realista, pues es el que emplean los sistemas modernos de memoria, y su implementación probablemente conduciría a mejoras de rendimiento. Sin embargo, no ha sido implementado en el modelo simplificado del controlador de memoria, que simplemente pretendía ser una versión sencilla que sirviera de soporte a la jerarquía de memoria.

- Evaluación de la propuesta asignando latencias al resto de instrucciones escalares. De nuevo, por simplicidad, solo se han tenido en cuenta latencias para las instrucciones de acceso a memoria y las instrucciones vectoriales de utilizadas por los benchmarks evaluados. A pesar de ello, para simular un modelo más fiel a la realidad, sería necesario contemplar la penalización de cualquier tipo de instrucción.

- Evaluación del rendimiento respecto del consumo energético de realizar los *lookups* cruzados y nativos en paralelo. Esta variante sería posible en

hardware, si bien conllevaría un indudable sobrecoste energético por lo que podría ser interesante explorar su idoneidad.

– Extensión del análisis para evaluar el rendimiento en aplicaciones vectoriales con patrón de acceso irregular por el uso de vectores indexados, es decir, con distinto *stride*. Estas podrían llegar a verse perjudicadas por el *prefetching* e, incluso, requerir una segregación de datos adicional en función del tipo de *stride* de sus referencias, de forma que los de accesos vectoriales indexados compartan cache con los de accesos escalares, evitando que interfieran con los datos vectoriales accedidos secuenciales.

Bibliografía

[1] V. Porpodas y T. M. Jones, «Throttling Automatic Vectorization: When Less Is More», *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 432-444, 2015.

[2] S. Rebolledo, «Introducción a la evaluación de la Arquitectura SVE en gem5», Universidad de Cantabria, 2022.

[3] S. G. Berg, «Cache Prefetching. Technical Report UW-CSE 02-02-04», University of Washington, Seattle, 2002.

[4] P. Hsu, «GitHub», [En línea]. Disponible en: https://github.com/phaa-eu/cavatools. [Último acceso: febrero 2024].

[5] J. L. Hennessy y D. A. Patterson, Computer Architecture: A Quantitative Approach, 1990.

[6] ARM Developer, «Scalable Vector Extension (SVE)», [En línea]. Disponible en: https://developer.arm.com/Architectures/Scalable%20Vector%20Extensions.

[7] Y. Zheng, B. T. Davis y M. Jordan, «Performance evaluation of exclusive cache hierarchies», *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004.*, pp. 89-96, 2004.

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

[9] S. Harris y D. Harris, Digital Design and Computer Architecture, 2015.

[10] W. Stallings, Computer Organization and Architecture, 2002.

[11] O. Mutlu y T. Moscibroda, «Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors», *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 146-160, 2007.

[12] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson y J. D. Owens, «Memory Access Scheduling», *Proceedings of the 27th Annual International Symposium on Computer Architecture*, vol. 28, nº 2, p. 128–138, 2000.

[13] N. Li, W.-Y. Jiang, B. Wu y K. Cao, «Improve DRAM Leakage Issue During RAS Operational Phase Through TCAD Simulation», *2019 IEEE 13th International Conference on ASIC (ASICON)*, pp. 1-4, 2019.

[14] RISC-V Foundation, «RISC-V», [En línea]. Disponible en: https://riscv.org/. [Último acceso: febrero 2024].

[15] D. A. Patterson, «Reduced Instruction Set Computers», *Association for Computing Machinery*, vol. 28, nº 1, 1985.

[16] RISC-V International, RISC-V 'V' Vector Extension Specification, 2021.

[17] R. F. Ibáñez, «Introduction to the RISC-V Vector Extension», de *2022 ACM Summer School on HPC and AI. Barcelona Supercomputing Center*, 2022.

[18] P. Hsu, «Cavatools», Barcelona Supertcomputing Center, [En línea]. Disponible en: www.bsc.es/research-and-development/software-and-apps/software-list/cavatools.

[19] C. J. Lee, O. Mutlu, V. Narasiman y Y. N. Patt, «Prefetch-Aware DRAM Controllers», *2008 41st IEEE/ACM International Symposium on Microarchitecture,* pp. 200-209, 2008.

[20] C. Batten, R. Krashinsky, S. Gerding y K. Asanovic, «Cache Refill/Access Decoupling for Vector Machines», *37th International Symposium on Microarchitecture (MICRO-37'04),* pp. 331-342, 2004.

[21] R. Espasa y M. Valero, «A victim cache for vector registers», *Proceedings of the 11th International Conference on Supercomputing,* p. 293–300, 1997.

[22] A. Musa, Y. Sato, T. Soga, R. Egawa, H. Takizawa, K. Okabe y H. Kobayashi, «Effects of MSHR and Prefetch Mechanisms on an On-Chip Cache of the Vector Architecture», *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications,* pp. 335-342, 2008.

[23] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez y A. Cristal, «A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures», *ACM Transactions on Architecture and Code Optimization,* vol. 17, nº 4, 2020.

[24] W. -C. Hsu y J. E. Smith, «Performance of cached DRAM organizations in vector supercomputers», *Proceedings of the 20th Annual International Symposium on Computer Architecture,* vol. 21, nº 2, p. 327–336, 1993.

[25] J. W. C. Fu y J. H. Patel, «Data Prefetching in Multiprocessor Vector Cache Memories», *[1991] Proceedings. The 18th Annual International Symposium on Computer Architecture,* pp. 54-63, 1991.

[26] J. B. Rothman y A. J. Smith, «Sector Cache Design and Performance», de *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728),* IEEE, 2000, pp. 124-133.

[27] R. Balasubramonian, Innovations in the Memory System. Synthesis Lectures on Computer Architecture (SLCA)., Springer Cham, 2022.

[28] C. Chen et al., «Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension: Industrial Product», *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA),* pp. 52-64, 2020.

[29] A. Barredo Ferreira, Novel Techniques to Improve the Performance and the Energy of Vector Architectures, Universitat Politècnica de Catalunya, 2021.