Facultad
de
Ciencias

Evaluación de políticas de reordenado de peticiones de memoria para cargas de trabajo sparse vectorizadas

(Evaluation of memory request reordering policies for vectorized sparse workloads)

Trabajo de Fin de Master
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Javier Canales García

Director: Borja Pérez Pavón

Septiembre-2025

Índice general

Índice de figuras

Índice de cuadros

Agradecimientos

En primer lugar, a mi familia, que siempre ha estado ahí para apoyarme independientemente de la situación y que, gracias a su trabajo, esfuerzo y cariño, me ha ayudado a llegar donde estoy tanto académicamente como en la vida.

A mis amigos, con los que paso una gran parte del tiempo y que siempre me muestran su apoyo y comprensión. Me proporcionan puntos de vista diferentes que me ayudan a crecer y dar una mejor versión de mí, tanto a nivel personal como profesional.

A mi pareja, esa persona con la que convivo prácticamente a diario, siempre me da mi lugar y me proporciona la calma y desconexión tan necesarias para mí. Al estar juntos, los problemas quedan atrás y nos retroalimentamos para hacernos el uno al otro mejor persona.

A los profesores que he tenido durante toda mi vida académica, que me transmitieron su conocimiento y pasión por querer aprender cosas nuevas día a día.

Y por último, a Borja y a José Luis, que accedieron a trabajar conmigo de nuevo para el desarrollo de este TFM al igual que ocurrió con el TFG, y que me han guiado semana a semana para avanzar y finalizar satisfactoriamente este proyecto.

Resumen

La sociedad actual demanda cada vez más y más rendimiento de los diferentes dispositivos, pero, sin embargo, se están alcanzando cotas tan altas que estas continuas mejoras no serán capaces de mantener el ritmo. Es por ello que se requiere exprimir al máximo los diferentes componentes para conseguir cualquier mejora que sea posible. En el caso de este estudio, se trabajará sobre el controlador de memoria de los procesadores, estudiando el efecto que tiene el reordenado de las peticiones a memoria a través de diferentes algoritmos de planificación, centrándonos principalmente en cargas de trabajo de tipo *sparse* vectorizadas.

El orden en el que se atienden las peticiones a memoria es uno de los factores clave en el rendimiento de las tecnologías basadas en *DRAM*. El ejemplo más sencillo es el aprovechamiento de la localidad en el *Row Buffer*: accesos sucesivos a la misma fila de un banco generalmente producirán anchos de banda superiores a cambios de fila constantes. Sin embargo, una priorización excesiva de accesos a una misma fila también puede tener un impacto negativo sobre el *fairness*, retrasándose excesivamente aquellos accesos que no aprovechan la localidad. Por este motivo, existen multitud de propuestas de reordenado de peticiones que tratan de resolver estos dos problemas contrapuestos.

Los sistemas actuales ejercen cada vez mayor presión sobre el sistema de memoria debido fundamentalmente a dos motivos: la integración de grandes cantidades de cores y la capacidad de ejecución SIMD o vectorial. A esto se le añade el florecimiento de aplicaciones de tipo *sparse*, las cuales, de forma natural, generan patrones de acceso muy desafiantes para el sistema de memoria.

En este contexto, este trabajo estudiará la interacción entre estos factores que incrementan la presión sobre la memoria y el reordenado de peticiones. Para ello, se implementarán varias políticas de reordenado en el simulador arquitectural de las ISA RISC-V Coyote y se evaluarán en la ejecución de un SpMV utilizando matrices con patrones de irregularidad variados.

Palabras clave: Simulador, Coyote, FRFCFS, BLISS, STFM, Algoritmo de reordenado, Planificación de peticiones de memoria

Abstract

Today's society demands ever-increasing performance from different devices, yet such high levels are being reached that these continuous improvements will not be able to keep pace. Therefore, it is necessary to squeeze the maximum out of the different components to achieve any possible improvement. In the case of this study, we will work on the processors' memory controller, studying the effect of reordering memory requests through different scheduling algorithms, focusing primarily on vectorized sparse workloads.

The order in which memory requests are served is one of the key factors in the performance of DRAM-based technologies. The simplest example is the exploitation of locality in the Row Buffer: successive accesses to the same row in a bank will generally produce higher bandwidths than constant row changes. However, excessive prioritization of accesses to the same row can also have a negative impact on fairness, with accesses that do not exploit locality being excessively delayed. For this reason, there are many request reordering proposals that attempt to solve these two opposing problems.

Current systems are putting increasing pressure on the memory system primarily for two reasons: the integration of large numbers of cores and the ability to execute SIMD or vector-based applications. Added to this is the rise of sparse applications (big data, machine learning, etc.), which naturally generate access patterns that are very challenging for the memory system.

In this context, this work will study the interaction between these factors that increase memory pressure and request reordering. To this end, several reordering policies will be implemented in the Coyote RISC-V ISA architectural simulator and evaluated in the execution of a SpMV using matrices with varied irregularity patterns.

Keywords: Simulator, Coyote, FRFCFS, BLISS, STFM, Reordering algorithm, Memory Requests Scheduling

Capítulo 1

Introducción

En este capítulo se recogerán las líneas base que dieron pie al desarrollo de este proyecto. Se comenzará indicando los motivos históricos que nos han llevado a tener los sistemas actuales tal y como los conocemos, así como la importancia de los simuladores en la investigación de nuevas áreas de mejora en el mundo de la computación. Además, se describirán los objetivos que se pretenden conseguir tras la realización del proyecto, el plan de trabajo seguido para su realización y la estructura del documento.

1.1. Visión tradicional de la memoria

La concepción tradicional que se tiene de un computador es la conocida como arquitectura Von Neumann, en la que se pueden distinguir principalmente 3 componentes: CPU, memoria y periféricos, es decir, dispositivos de entrada/salida. Ese diseño resulta funcional, pero en la actualidad no es suficiente. La sociedad en la que vivimos exige cada vez más y más rendimiento y estas altas cotas requieren de innovaciones que nos permitan realizar los cómputos de forma más rápida y eficiente. Con el paso de los años, el componente que ha acaparado prácticamente todas las mejoras es la CPU, pasando del procesador monociclo al superescalar, al fuera de orden, al multicore, etc. Por su parte, la memoria también ha ido mejorando, pero no al mismo ritmo al que lo han hecho los procesadores [1]. En este componente podemos hacer la distinción entre la capacidad de almacenamiento y el rendimiento (velocidad de acceso a los datos).

Mientras que la capacidad ha mejorado enormemente con el paso de los años, el rendimiento no ha seguido los mismos pasos. En poco tiempo se ha conseguido pasar de una capacidad del orden de los KB, disponibles en los primeros ordenadores, a los sistemas actuales que cuentan con capacidades del orden de los GB, llegando incluso a TB en entornos HPC. Este aumento significativo sí que es acorde a las mejoras realizadas a la CPU, pero el rendimiento no ha evolucionado de la misma forma, y esto se debe principalmente al tipo de tecnologías

empleadas en su construcción.

La tecnología habitual empleada en la construcción de la memoria es la *DRAM*, en la que por cada bit se emplea un condensador que almacena la información, pero que pierde la carga de forma progresiva, por lo que es necesario un refresco para conservar el valor.

La principal alternativa a la tecnología *DRAM* es la *SRAM*. Esta se diferencia de la anterior en que para almacenar un bit se emplea un sistema con dos inversores que forman un bucle, y este conserva el valor sin la necesidad de refresco. Esto supone una ventaja, ya que las latencias son menores, pero la densidad resultante es mucho menor en comparación con la *DRAM*, ya que aquí se necesitan seis transistores por bit y con la *DRAM* únicamente un transistor y un condensador.

En la figura 1.1 se pueden observar las diferencias en la construcción de una celda *DRAM* y una *SRAM*.

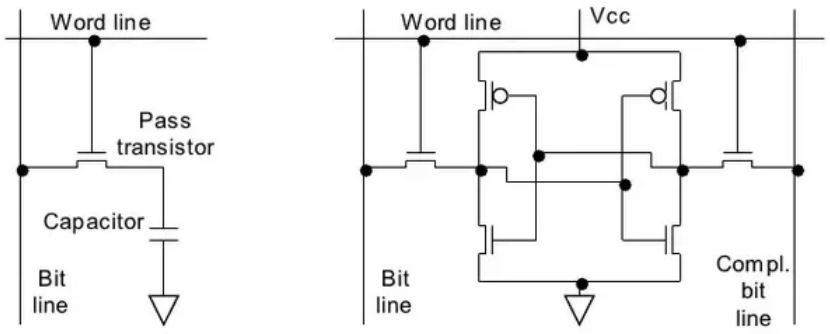

Figura 1.1: DRAM vs SRAM

Las razones detrás de por qué la tecnología *DRAM* es la principal tecnología empleada en la construcción de memorias, residen en que para los chips de memoria se ha priorizado la densidad sobre la latencia. Para un mismo espacio, la capacidad resultante al emplear celdas *DRAM* es muy superior a la obtenida con las celdas *SRAM*. Además, debido a la tecnología subyacente, el coste de fabricación de las memorias *DRAM* es inferior al de las *SRAM* por lo que el precio de acceso a ellas será más beneficioso para los usuarios. Por último, mencionar que, debido a la jerarquía de memoria, a este nivel no prima tanto la velocidad, ya que existen mecanismos para minimizar las latencias, por lo que el gran punto fuerte de las memorias *SRAM* se ve ensombrecido [2]. Al no poderse emplear memoria de tipo *SRAM* para la memoria principal debido a los inconvenientes mencionados, se emplean mecanismos como las caches para compensar la diferencia de rendimiento respecto a la *DRAM*.

Esto ha dado lugar a lo que se conoce actualmente como "Memory Wall", que indica la limitación de rendimiento de los sistemas por parte de la memoria, ya que el procesador es capaz de ir varios órdenes de magnitud más rápido. Esta situación se agrava aún más en los últimos años con la introducción de los multicores, sistemas con múltiples cores en un solo chip. Este tipo de tecnología incrementa enormemente el tráfico entre la CPU y la memoria, haciendo que la diferencia de rendimiento sea aún más notable. El resultado es

que el rendimiento global del sistema se ve lastrado por la memoria, a pesar de las grandes mejoras realizadas en los procesadores.

En la figura 1.2 se puede observar cómo ha ido evolucionando el rendimiento tanto del procesador como de la memoria. En los primeros años, las mejoras iban a la par, pero con el paso del tiempo fue el procesador el que acaparó la mayoría de las mejoras, dejando a la memoria en un segundo plano. Esto no quiere decir que en el campo de la memoria no se hayan realizado esfuerzos, sino que los resultados obtenidos no han sido tan buenos como las mejoras propuestas en los procesadores.

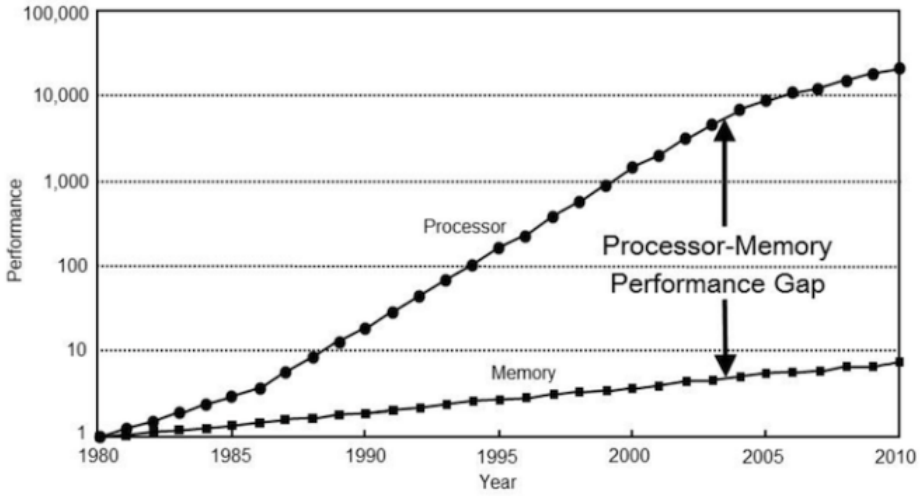

Figura 1.2: Memory Wall

Otro problema que agrava esta situación son las nuevas aplicaciones, Big Data, análisis de grafos, IAs, etc; que están tan de moda hoy en día. Este tipo de aplicaciones emplean cantidades ingentes de datos para funcionar de forma adecuada, por lo que el hardware empleado para procesarlos debe ser capaz de responder de forma eficiente. En muchas ocasiones, los tipos de datos empleados son de tipo *sparse*, lo que significa que los datos que aportan valor están separados entre sí y predominan los datos 0 o nulos. [3] Esto supone un problema aún mayor, ya que impide que los mecanismos para mitigar las latencias resulten útiles, como por ejemplo las caches, que se aprovechan de la localidad temporal de la información.

Las memorias caches son una de las principales tecnologías empleadas para reducir la diferencia de rendimiento entre el procesador y la memoria. Se implementan a través de la tecnología *SRAM* (poca capacidad, alto rendimiento), situándose cerca del procesador y, basándose en la localidad de los datos, permiten que su acceso sea de la forma más rápida posible. A partir de esta tecnología se desarrolla la jerarquía de memoria del ordenador, en la que, a medida que se va subiendo de nivel, la capacidad va aumentando y el rendimiento disminuyendo. De esta forma, las memorias cercanas al procesador son muy rápidas pero solo permiten almacenar unos pocos datos, y según nos vamos acercando a la memoria principal, estas características varían de forma inversamente proporcional. [2].

Además, la tecnología *DRAM*, empleada para construir la memoria principal, ha ido evo-

lucionando a lo largo de los años a través de diferentes versiones. Una de las mejoras más trascendentales es la tecnología *DDR*, que permite duplicar la velocidad de transferencia de datos en las memorias. Esto es así debido a que se aprovechan tanto los flancos de subida del reloj como los de bajada para realizar las transferencias, a diferencia de la tecnología usada hasta el momento, que era *SDR*, la cual únicamente permitía realizar las transferencias en los ciclos de subida. La diferencia entre estas dos tecnologías se puede observar en la imagen 1.3, teniendo la posibilidad de aprovechar cada uno de los flancos del reloj. *DDR* ha ido actualizándose con el paso del tiempo (DDR2, DDR3, etc) hasta llegar a *DDR6*, que es la tecnología actual. La principal diferencia entre las versiones es la velocidad del reloj y el ancho de banda resultante, pasando de valores inferiores a 1 Gb/s en las primeras versiones, hasta aproximadamente 135 GB/s de *DDR6*.

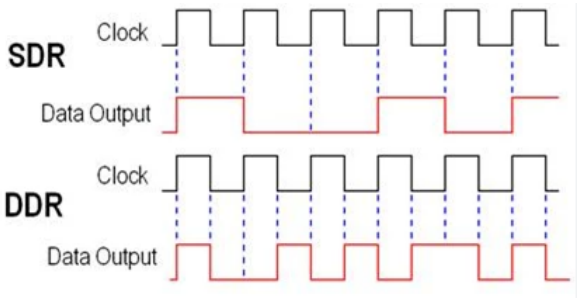

Figura 1.3: SDR vs DDR

El mundo de la memoria siempre ha resultado más costoso y cambiante, por lo que las empresas, a pesar de dedicar una buena parte de sus esfuerzos y recursos a la investigación en este campo, no han obtenido los resultados esperados, pero en los últimos tiempos esto ha cambiado. Aunque desde el punto de vista de la industria esto sigue así, ha surgido una corriente en el mundo académico que ha llevado a los investigadores a explorar nuevas y complejas ideas para maximizar el rendimiento de los sistemas. En muchas ocasiones, estas ideas trascienden y de forma inherente conllevan cambios en la industria en una mayor o menor medida.

A día de hoy, son muchos los avances que se pueden observar en los sistemas de memoria y que han dado lugar a nuevas ramas que antes no habían sido consideradas. Se han desarrollado productos radicalmente diferentes a la visión que se tenía en los inicios de la memoria, como pueden ser las HBM (High Bandwith Memory) [4] o los HMC (Hybrid Memory Cube) [5]. También se ha considerado dotar de inteligencia a la memoria y que no sirva únicamente para el almacenamiento de información, sino que sea capaz de realizar ciertas tareas que permitan liberar de carga a la CPU. Este campo se conoce como *Processing in Memory* o *Near Data Processing* [6].

Todas estas mejoras han contribuido enormemente a mejorar el rendimiento de los equipos, especialmente en el entorno del propósito general. Pero, a pesar de ello, existen situaciones en las que se siguen produciendo cuellos de botella en los sistemas de memoria, como pueden ser las aplicaciones dispersas con patrones de acceso a memoria que no aprovechan bien la jerarquía.

En este contexto, nos planteamos estudiar el problema de la planificación de peticiones de acceso a memoria de múltiples cores, dentro del controlador de memoria, para mejorar los tiempos de acceso teniendo en cuenta el modo de funcionamiento de la tecnología *DRAM*.

1.2. Objetivos

Este proyecto tiene como propósito fundamental el estudio de propuestas de planificación de peticiones de acceso a memoria en entornos multicore basados en arquitectura *RISC-V* vectorial. Para ello, se implementarán una serie de algoritmos de reordenado de peticiones de memoria y se compararán para determinar cuál consigue un mayor rendimiento en función del escenario de ejecución. De esta forma, el trabajo se desglosa en tres bloques principales:

- *Selección de algoritmos de planificación*: Muchos sistemas optan por emplear algoritmos básicos a la hora de realizar la planificación de peticiones de acceso a memoria debido a su sencillez de implementación, pero es conocido que estos no resultan los más eficientes. Se tratará de seleccionar una serie de algoritmos de planificación que puedan mejorar su rendimiento, especialmente en aplicaciones con patrones de acceso *sparse*.

- *Implementación de los algoritmos seleccionados en Coyote*: Se empleará el entorno de simulación *Coyote* para implementar los diferentes algoritmos de planificación seleccionados y evaluar su rendimiento. Antes de realizar la comparativa de rendimiento se deberá validar que el comportamiento de los algoritmos es el esperado, por lo que se incluirán diferentes métricas que nos permitirán determinar si el funcionamiento es correcto.

- *Estudio del impacto de los algoritmos en la herramienta*: Diferentes algoritmos pueden tener como objetivos diferentes métricas, por lo que el estudio se centrará principalmente en el tiempo de ejecución, la ocupación del sistema (saturación del sistema de memoria) y el *fairness* en la ejecución de los diferentes *threads*. Por otro lado, se estudiará la escalabilidad de los algoritmos, analizando su comportamiento a medida que se aumentan los recursos simulados. Para la evaluación se utilizarán distintas configuraciones de hardware y una batería de matrices con datos de tipo *sparse* para ejecutar una aplicación de tipo matriz por vector.

1.3. Plan de trabajo

Para la consecución de los objetivos descritos en la Sección 1.2 se ha seguido el siguiente plan de trabajo:

- *Análisis de políticas de planificación*: El primer paso a seguir para la realización de este trabajo es conocer cuál es el problema que debemos tratar y proponer las soluciones adecuadas. Para ello se estudiarán una serie de artículos relacionados con la planificación de peticiones de memoria y se seleccionarán aquellos que nos resulten más interesantes de implementar en función de diferentes métricas.

- *Análisis de Coyote*: Antes de comenzar a trabajar y modificar la herramienta es necesario conocer cómo funciona para así ser consciente de los elementos a modificar de forma efectiva. El simulador dispone tanto de documentación como de papers asociados. [7]

- *Adecuación de los parámetros de entrada del simulador*: Por la forma en la que está construido Coyote, es necesario que los datos de entrada se encuentren en ficheros de cabecera de C. Esto se debe a que el sistema es *baremetal* y por tanto no puede hacer uso de llamadas al sistema para facilitar la lectura de ficheros. La batería de matrices [8] que se utilizarán para los benchmarks se encuentran en formato comprimido, por lo que es necesario realizar la transformación de los datos.

- *Implementación de los algoritmos seleccionados en el primer punto*: De todos los algoritmos analizados se decide que los más apropiados para implementar son tres. El primero de ellos es *FRFCFS* que supone una evolución sobre *FIFO*, permitiendo el adelantamiento de peticiones [9]. El segundo es *BLISS* que trata de equilibrar el rendimiento y el *fairness* [10]. Y el último es *STFM* que resulta el más complejo y emplea heurística para determinar cuál es la siguiente petición a planificar [11]. Todos ellos serán implementados sobre el simulador, se validará su funcionamiento y se realizará una batería de pruebas para contrastar el rendimiento obtenido.

- *Análisis de la paralelización del programa empleado*: Se estudiará como se realiza el reparto de trabajo entre los diferentes cores y se tratará de optimizar para intentar sacar el máximo rendimiento posible.

- *Comparativa de algoritmos*: Una vez tenemos los algoritmos correctamente implementados y depurados, se compararán sus rendimientos para conocer cual funciona mejor en determinadas situaciones.

1.4. Estructura del Documento

El documento elaborado consta de cinco capítulos, incluyendo el presente *Capítulo 1* de *Introducción*.

- *Capítulo 2*: Este capítulo permite conocer con un mayor detalle los diferentes elementos que serán tratados en el proyecto. Se hará una descripción de la herramienta a utilizar, así como de los diferentes algoritmos a implementar.

- *Capítulo 3*: En este capítulo se detallarán las modificaciones que se han llevado a cabo en el simulador para conseguir implementar los algoritmos mencionados. Se razonarán las decisiones de diseño seguidas y los problemas encontrados en el camino.
- *Capítulo 4*: Se presentarán los experimentos a realizar para comprobar que, por un lado los algoritmos están correctamente implementados, y por otro lado, el resultado obtenido en términos de rendimiento es el esperado. Los datos de prueba son públicos, por lo que cualquier persona puede utilizarlos para replicar el estudio y obtener los mismos resultados.
- *Capítulo 5*: Para concluir, se hará un repaso de los objetivos planteados, comprobando si estos han sido cumplidos, y se indicarán las posibles vías de mejora del simulador Coyote.

Capítulo 2

Background

Este capítulo se emplea para introducir los conceptos más importantes sobre los que se basará el resto del proyecto. En primer lugar, se describen una serie de términos importantes relacionados con el trabajo. A continuación, se hará una breve explicación del funcionamiento de la memoria principal y una pequeña sección indicando las características principales de la arquitectura *RISC-V* y las instrucciones vectoriales. Además, se realizará una explicación somera del simulador utilizado en el trabajo (Coyote), su funcionamiento y cómo está implementado. Y, por último, se muestran una serie de trabajos de otros autores que tienen una estrecha relación con el desarrollo de este.

2.1. Glosario

Esta sección sirve para indicar algunos de los términos clave que se emplearán en el desarrollo del trabajo, lo cual facilitará su comprensión e interpretación. Se trata tanto de conceptos inherentes a la planificación de peticiones en el controlador de memoria, como de términos básicos del sistema de memoria, o siglas habituales que aparecen a lo largo de los diferentes capítulos.

- *Controlador de memoria*: Elemento del ordenador encargado de gestionar la comunicación entre el procesador y la memoria. Entre sus principales tareas están realizar la lectura y escritura en la memoria *DRAM*, y controlar sus intervalos de refresco. Puede ser independiente o venir integrado en el propio procesador.
- *Política de planificación*: Conjunto de estrategias y algoritmos empleados en el controlador de memoria para gestionar el acceso a la memoria principal por parte de múltiples peticiones provenientes del procesador.
- *Fairness*: Característica del sistema que indica si los recursos del sistema son tratados

de forma justa o, por el contrario, determinados elementos se ven beneficiados en favor de otros.

- *RAS*: Siglas de *Row Address Strobe*: Comando lanzado en el sistema de memoria que indica la linea a activar.
- *CAS*: Siglas de *Column Address Strobe*: Comando lanzado en el sistema de memoria que indica la columna a activar.
- *Hilo / Thread*: Unidad más pequeña de procesamiento que puede ser gestionada por el sistema operativo. Secuencia de tareas que puede ser ejecutada por diferentes procesadores de manera concurrente. Un proceso puede tener múltiples hilos y estos comparten recursos.

2.2. Memoria Principal

La memoria principal de un ordenador está basada en memoria *DRAM*, que organiza los datos en una jerarquía específica para tratar de maximizar la eficiencia en el acceso.

En cuanto a su estructura, la memoria se va descomponiendo de forma jerárquica hasta llegar a las celdas, que son las encargadas de almacenar los datos [12]. Si analizamos un módulo, estos serían los principales componentes:

- *Módulo de memoria (DIMM)*: Se trata de una *PCB* que contiene chips de memoria en ambas caras. El consorcio *JEDEC* define el estándar *DDR*, que es seguido por la gran mayoría de fabricantes (Intel, IBM, AMD, etc) para permitir la interoperabilidad en sus productos.
- *Canal de memoria*: Es la vía que conecta el controlador de memoria con los módulo *DIMM*. Si coexisten varios canales en el mismo sistema, estos pueden ser independientes o en paralelo (dual-channel, quad-channel, etc) para obtener un mejor rendimiento.
- *Rank*: Un rank es una colección de chips *DRAM* que trabajan de forma cooperativa para mantener el bus de datos ocupado en cada ciclo. En función del tamaño del bus, la organización de los chips se hará de una forma u otra. Por ejemplo, si el bus tiene un tamaño de 64 bits (en *DDR*, *DDR2* o *DDR3*) y cada chip tiene un ancho de 8 bits, serán necesarios 8 chips para conformar un rank, mientras que si el ancho de cada chip es de 4 bits, serán necesarios 16.
- *Bank*: Es un set de arrays independientes dentro de un chip *DRAM*. Son unidades de almacenamiento independientes que permiten accesos concurrentes. Pueden trabajar en una petición diferente de forma aislada, lo que proporciona un mayor nivel de paralelismo y maximiza las oportunidades de que haya datos disponibles para transferir al bus en cada ciclo. Los datos dentro del Bank se almacenan en una matriz bidimensional, y se acceden a través su fila y columna.

- *Row Buffer*: Se trata de un buffer existente para cada bank que almacena la última fila accedida. Su uso es clave para la eficiencia de acceso.

En la figura 2.1 se pueden observar los diferentes elementos que componen una memoria *DRAM*:

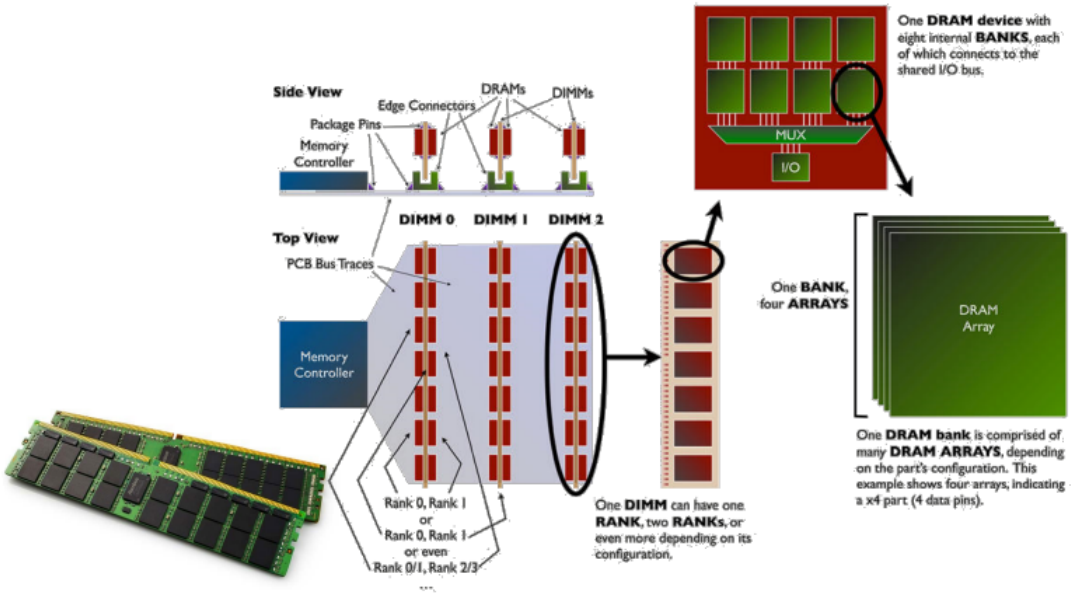

Figura 2.1: Componentes DRAM

A través de la dirección de la petición, se determina cuáles son los elementos accedidos del sistema de memoria (fila, banco, columna). Están codificadas de tal forma que cada subgrupo de bits que componen la dirección completa, representa un lugar específico y, todas ellas juntas, permiten acceder a un dato en concreto. Por ejemplo, con direcciones de 32 bits, se pueden codificar hasta 2^{32} direcciones, siendo un ejemplo de descomposición 2 bits para el banco, 14 bits para la fila y 16 bits para la columna.

La CPU lanza la petición con la dirección y esta es recibida por el controlador de memoria. Este elemento se encarga de transformar la solicitud de datos por parte del procesador en comandos que son comprensibles para el sistema de memoria. Establece el canal, banco, fila y columna necesarios para obtener la información solicitada. Además, cuenta con políticas que permiten priorizar las peticiones para tratar de maximizar el rendimiento.

Teniendo en cuenta esta descomposición de las peticiones en comandos del sistema de memoria, los principales a tener en cuenta son los siguientes:

- *ACT*: Comando encargado de cargar una fila desde las celdas de memoria hasta el *Row*

Buffer. Es necesario ejecutarlo antes de poder leer o escribir una fila. Tiene asociado el tiempo tRAS.

- *READ/WRITE*: Estos comandos permiten acceder a los datos almacenados en la memoria, a través del *Row Buffer*. En el caso de la lectura, lee los datos de la columna especificada y los deposita en el bus de memoria, mientras que la escritura se hace de forma *buffereada*, los datos del bus se escriben en la columna específica del *Row Buffer* y luego estos son transferidos a las celdas de memoria. Tiene asociado el tiempo tCAS.
- *PRE*: En el caso de querer leer o escribir una columna de una fila, puede ocurrir que exista otra fila cargada en el *Row Buffer* por lo que se produce un conflicto. La fila que actualmente se encuentra en el *Row Buffer* debe ser guardada en las celdas correspondientes de memoria, antes de realizar la carga de la nueva fila. Tiene asociado un tiempo tPRE.
- *REF*: El controlador de memoria es el encargado de gestionar el refresco de las celdas para evitar la pérdida de datos, por lo que cada cierto tiempo (64 ms en la mayoría de memorias *DRAM* [13]) se envía este comando para recargar los condensadores que componen las celdas.

La lectura o escritura de la información no se puede hacer de forma arbitraria en las diferentes celdas de la memoria, sino que todo debe pasar por el *Row Buffer*. Cuando se quiere hacer una lectura, la fila seleccionada de la memoria es cargada en el *Row Buffer* y desde él se leen los datos que van a ser enviados de vuelta al procesador. Si por el contrario lo que se quiere realizar es una escritura, se carga la fila en el *Row Buffer*, se modifican los datos ahí, y se realiza la escritura del *Row Buffer* en las celdas de memoria.

Este elemento resulta de vital importancia para el rendimiento de la memoria. Uno de los algoritmos seleccionados para la realización de este trabajo, *FRFCFS*, tiene como objetivo maximizar el número de aciertos que se producen en el *Row Buffer* y de esta forma obtener un mayor rendimiento. Al conseguir un mayor número de *hits* en el *Row Buffer*, se reduce el número de cambios de línea, por lo que el tiempo de ejecución es menor. Esto se debe a los tiempos asociados para realizar cada una de las acciones en el sistema de memoria:

- En el caso de producirse un acierto en el *Row Buffer*, unicamente se necesita hacer una operación de lectura/escritura en la columna deseada, por lo que el tiempo resultante sería unicamente tCAS. Este escenario es el más deseable ya que se reutiliza la fila existente.
- Si se trata de acceder a una fila, y esta no está en el *Row Buffer*, se necesita realizar una apertura previa, por lo que se debe añadir un tiempo adicional tRAS. Una vez se tiene la fila en el *Row Buffer*, se sigue el mismo flujo que en el caso de acierto, añadiendo el tiempo tCAS.
- El caso más desfavorable que puede producirse, ocurre cuando se intenta leer una fila del *Row Buffer* pero ya hay otra fila que previamente debe escribirse en su correspondiente

lugar de la memoria. En este caso el tiempo total estaría compuesto por el tiempo de almacenamiento tPRE, el tiempo de carga de la fila deseada en el *Row Buffer* tRAS, y el tiempo propio de acceso al dato tCAS.

Se está empleando un modelado simplificado de la memoria ya que, en la realidad, existen interacciones más complejas y con más parámetros temporales que dificultarían la comprensión y los cálculos. Aun así, este modelado nos permite obtener resultados relevantes y precisos.

Los algoritmos de planificación seleccionados tratarán de obtener las mejoras de rendimiento a través de diferentes parámetros de la memoria, como puede ser la maximización de los aciertos en el *Row Buffer*, en el caso de *FRFCFS*, o adicionalmente mejorando el *fairness* entre *threads* con los algoritmos *BLISS* y *STFM*.

2.3. RISC-V e instrucciones vectoriales

La arquitectura de los ordenadores se divide principalmente en 2 grandes grupos, conocidos como *CISC* y *RISC*. Estos son 2 clases de *ISAs* (Instruction Set Architecture), es decir, conjuntos de instrucciones que determinan las operaciones a bajo nivel que puede realizar el ordenador.

La visión de *CISC* consiste en un gran conjunto de instrucciones complejas para poder realizar prácticamente cualquier operación con una sola instrucción. Es decir, son instrucciones potentes, que cuentan con una gran funcionalidad, pero que son más costosas de ejecutar. Esta ha sido la filosofía seguida, por ejemplo, por *Intel* a la hora de diseñar sus productos.

Y por otro lado, está la visión de *RISC*, que consiste en un grupo reducido de instrucciones, más sencillas y rápidas de ejecutar, pero que tienen una funcionalidad más limitada. Esta ha sido la filosofía seguida, por ejemplo, por *ARM*.

Para comprender mejor la diferencia entre estos 2 grupos, podemos suponer que queremos realizar una operación consistente en una multiplicación de enteros. En la arquitectura *CISC* solo necesitaríamos utilizar una única instrucción que se encargaría de la lectura de los operandos en los registros, la propia operación y la escritura del resultado. Sin embargo, empleando una arquitectura *RISC*, este proceso sería dividido en varias instrucciones; se realizarían los *LOADs* correspondientes, la operación y el *STORE*.

Dentro de *RISC*, existe una corriente que da lugar a *RISC-V*. Partiendo de las bases de *RISC*, destaca por ser abierta y libre de *royalties*, por lo que cualquier persona puede usarla, modificarla e implementarla sin pagar licencias. Gracias a esta característica, además de su potencia, hace que sea un área muy importante a investigar tanto para las empresas como a nivel académico.

Además de las mejoras en las arquitecturas y las distintas áreas relacionadas con los computadores, es posible exprimir la forma en la que se ejecutan las instrucciones. En este contexto surgen las instrucciones vectoriales, para conseguir un mayor rendimiento basándose en el paralelismo.

Tradicionalmente, las operaciones siempre se han realizado de forma secuencial, operando a operando, pero en los últimos tiempos se han incluido en las nuevas arquitecturas, gracias al soporte *hardware*, las instrucciones vectoriales, que son capaces de realizar operaciones sobre múltiples elementos en un solo ciclo.

Supongamos que tenemos 2 arrays A y B de 5 elementos cada uno (0-4) y queremos hacer, por ejemplo, la suma de sus elementos en otro array C. En una arquitectura tradicional, para realizar esta operación se necesitarían al menos 5 ciclos (A[0] + B[0] = C[0], A[1] + B[1] = C[1], ..., A[4] + B[4] = C[4]), mientras que, empleando una instrucción vectorial, este tipo de operaciones se podría realizar en un único ciclo (A[0:4] + B[0:4] = C[0:4]).

Este tipo de instrucciones se emplean, por ejemplo, en los procesadores de tipo *SIMD* (Single Instruction, Multiple Data) como pueden ser los *AVX* de *Intel* y *AMD*, o en las *GPUs* ya que aumenta el rendimiento en tareas con un volumen alto de datos repetitivos, además de reducir el consumo energético por operación al hacer más trabajo por ciclo.

2.4. Simulador Coyote

La creciente demanda de simulación y diseño hardware-software para supercomputadores de alto rendimiento (HPC) ha impulsado el desarrollo de herramientas abiertas que permitan la exploración y optimización arquitectónica desde las primeras etapas de diseño. En este contexto, el proyecto MEEP (MareNostrum Experimental Exascale Platform) del BSC (Barcelona Supercomputing Center) introduce Coyote, un simulador open source basado en la arquitectura RISC-V [7].

El proyecto MEEP tiene como foco abordar dos objetivos principalmente:

- Por un lado, la validación pre-silicio de la propiedad intelectual (IP) mediante emulación basada en FPGA. Esto resulta esencial en el desarrollo de avances en el ámbito del hardware, ya que se puede validar el comportamiento de las soluciones de manera ágil y precisa sin necesidad de la construcción real, disminuyendo tanto el tiempo de desarrollo como la utilización de recursos.

- Y por otro lado, el desarrollo de software para preparación de nuevas arquitecturas hardware. Resulta necesario desarrollar software capaz de adaptarse a los cambios que se vayan produciendo y, de esta forma, las simulaciones realizadas serán igual de eficientes y precisas.

MEEP propone ACME (Accelerated Compute and Memory Engine), una arquitectura *RISC-V* que reduce los cuellos de botella en HPC al integrar el procesamiento directamente en el acelerador. Con procesadores vectoriales y arrays sistólicos, ACME soporta tanto cargas tradicionales como nuevas aplicaciones de IA y análisis de datos, lo que impulsa la necesidad de simuladores eficientes como Coyote para explorar el diseño desde etapas tempranas.

Coyote es un simulador conducido por ejecución que se basa en dos herramientas preexistentes:

- *Spike*: Esta herramienta es el estándar número uno para la simulación de arquitecturas *RISC-V*, ya que es utilizada ampliamente por la comunidad. Es fundamentalmente un simulador funcional de las *ISA* sin apenas capacidades para modelado de rendimiento. Soporta muchas de las capacidades requeridas en el diseño de sistemas HPC, entre las que se encuentran las instrucciones vectoriales o los sistemas multicore. A pesar de contar con múltiples competencias en el ámbito cercano al chip (CPU y L1), tiene ciertas carencias a medida que se va alejando en la jerarquía de memoria; y aquí es donde entra la segunda de las herramientas.
- *Sparta*: Herramienta utilizada para construir modelos de rendimiento basados en eventos flexibles, con un diseño modular, encapsulando la funcionalidad de cada elemento en un componente. Permite diseñar jerarquías de memoria complejas, incluyendo elementos que van desde la L2 en adelante o la red de interconexión.

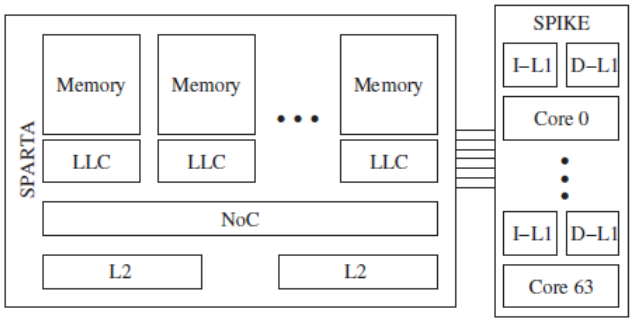

Figura 2.2: Arquitectura Coyote

El objetivo principal de Coyote es modelar el movimiento de datos a través de las jerarquías de memoria para identificar cuellos de botella en arquitecturas HPC. A través de la simulación, el usuario es capaz de determinar los elementos del sistema que provocan una pérdida de rendimiento, y, por tanto, deben ser analizados y rediseñados.

Coyote cuenta con múltiples ventajas, entre las que destacan su flexibilidad y modularidad, permitiendo ajustar configuraciones de memoria, como tamaño de caché, políticas de mapeo y latencias; su velocidad, ya que alcanza un rendimiento de hasta 6 millones de instrucciones por segundo (MIPS) simulando sistemas de hasta 128 núcleos, permitiendo comparaciones rápidas de diferentes diseños; y su compatibilidad, soportando instrucciones vectoriales RISC-V y cargas de trabajo paralelas, esenciales para HPC.

Coyote por sí solo ofrece buenos resultados, pero si se integra con otras herramientas, las posibilidades aumentan enormemente. Por ejemplo, es posible emplear Paraver (herramienta para el análisis de rendimiento) con los resultados obtenidos tras la simulación y, de esta forma, ser capaces de determinar de una forma más sencilla dónde se están produciendo los cuellos de botella en la aplicación estudiada. Al ser desarrollado siguiendo la filosofía open source, se promueve su reutilización en otros proyectos de la comunidad RISC-V, lo que contribuye a un ecosistema abierto que abarca desde aplicaciones hasta hardware.

La utilización de Coyote permite a desarrolladores y diseñadores de hardware evaluar nuevos diseños antes de su implementación en una FPGA, lo que acelera el ciclo de desarrollo y, a largo plazo, la utilización de herramientas como esta será fundamental para el avance de arquitecturas Exascale (sistemas de computación capaces de realizar un mínimo de 1 exaflop por segundo) y la consolidación de RISC-V en HPC. Además, al integrarse en el ecosistema de MEEP, contribuye al desarrollo de herramientas abiertas para supercomputadores de próxima generación y fomenta la innovación en el campo de la computación de alto rendimiento.

2.5. Trabajos Relacionados

El problema de planificación, independientemente del contexto en el que se emplee, es NP-completo, lo que significa que no existe un algoritmo que sea el idóneo para utilizar en todas las situaciones, por lo que se deben emplear algoritmos heurísticos para tratar de dar solución a este problema. En función de cuál sea la métrica objetivo, se tomará una elección de algoritmo u otra, ya que no existe una opción que sea capaz de conseguir el mejor resultado para todos los parámetros. Un algoritmo que trate de maximizar el rendimiento para una métrica, tendrá alguna deficiencia en otra, que a su vez será subsanada por otro algoritmo. [14].

Además, centrándonos en la planificación de peticiones a memoria, este problema se agravará en el futuro, ya que se diseñarán nuevos procesadores con una mayor cantidad de núcleos y la compartición de las caches hará que se sature aún más el sistema de memoria. El rendimiento de las cargas de trabajo varía en función del rendimiento de las aplicaciones en la LLC, ya que es compartida por múltiples cores. En el trabajo de Feliu et al. [15], se propone el estudio de este elemento para determinar cómo influye la geometría de la cache(número de conjuntos, número de vías y tamaño de la línea) y arquitectura en el rendimiento de las aplicaciones. Adicionalmente, se propone un algoritmo de planificación que se basa en el ancho de banda disponible en cada nivel de la jerarquía para hacer la priorización de peticiones. Selecciona las peticiones de tal forma que se minimicen los efectos de contención en memoria.

Las mejoras en el controlador de memoria no tienen por qué venir únicamente dadas a través del planificador empleado. Existen otros elementos dentro del controlador que también pueden ser objeto de estudio para obtener un mayor rendimiento de los sistemas. En el trabajo *Diseño de controladores de memoria eficientes para futuros sistemas* [16] se exponen

una serie de técnicas para tratar de reducir el tiempo de acceso, como por ejemplo, emplear técnicas de pre-búsqueda en sistemas multicore.

El hecho de buscar continuamente mejoras en el rendimiento de los sistemas es aplicable a cualquier nivel, por lo que para cada uno de los elementos que los componen se están buscando novedosas formas de optimizarlo para conseguir mejoras independientemente de la métrica objetivo. Por ejemplo, en el artículo *Effect of context aware scheduler on TLB* [17], se propone un algoritmo de planificación cuyo objetivo es tratar de reducir el número de vaciados que se realizan en el TLB (Translation Lookaside Buffer) para intentar agilizar los accesos. Al realizar un cambio de contexto y, por tanto, un cambio en el espacio de direcciones, en algunos procesadores se realiza un vaciado del TLB, lo que implica rellenarlo de nuevo con nuevas traducciones. En el estudio demuestran que el algoritmo propuesto consigue reducir el número de fallos en el *TLB* e incrementar el rendimiento del sistema.

En la actualidad, se están desarrollando grandes mejoras en los campos de la inteligencia artificial y el aprendizaje automático, buscando introducirlas en prácticamente cualquier elemento. A nivel del controlador de memoria, gracias a los avances *hardware* producidos en los últimos años, también es posible introducir mejoras relacionadas con este campo, como proponen Ipek et al. [18]. Los algoritmos empleados tradicionalmente son estáticos, es decir, no cambian su comportamiento independientemente de la carga que estén manejando, lo que los hace poco flexibles, pero la tendencia actual consiste en diseñar algoritmos dinámicos que modifiquen su comportamiento en función del estado del sistema (como es el caso del algoritmo *STFM*, implementado en este trabajo [11], que realiza la selección de peticiones en función del *slowdown*). Se propone un diseño de controlador de memoria que, basándose en aprendizaje reforzado (RL), observa el estado del sistema y se adapta para garantizar que la acción tomada es la más ventajosa a largo plazo. Los resultados muestran que el ancho de banda resultante es un 22 % mayor que el obtenido por los controladores de memoria tradicionales.

Además, los diferentes elementos que componen el sistema han sido diseñados de acuerdo a la tecnología existente en la época. No se puede tratar de emplear algoritmos y técnicas en los sistemas actuales que eran eficientes hace 30 años, ya que la tecnología ha evolucionado y en la actualidad se dispone de componentes más potentes. En el documento *ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers* [19], los autores muestran una técnica de planificación denominada ATLAS que permite aumentar el rendimiento del sistema en entornos multi-core con varios controladores de memoria. La idea que reside detrás de esta técnica es priorizar los *threads* en función del controlador de memoria al que están asociados, reduciendo el tiempo que estos están bloqueados. Tras realizar pruebas con diferentes cargas de trabajo y sistemas, se determina que el rendimiento mejora un 8,4 % en comparación con otros algoritmos tradicionales diseñados para sistemas multi-core.

Y por último, adicionalmente a los algoritmos seleccionados para su implementación en este trabajo, se valoraron otras posibilidades que fueron descartadas, pero cuya implementación queda abierta para realizar en sucesivos trabajos. Un ejemplo es el algoritmo PAR-BS des-

crito en el artículo *Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems* [20], donde se propone un controlador para sistemas multiprocesador con memoria *DRAM* compartida. Proporciona calidad de servicio a los *threads* y mejora el rendimiento del sistema mediante el procesado de peticiones en bloques. Además, se optimiza el acceso a los bancos de memoria a través del paralelismo, reduciendo el tiempo de bloqueo de los *threads* debido a la memoria. Las pruebas realizadas indican que el rendimiento obtenido es un 8,3 % superior frente a otros algoritmos de referencia. Este algoritmo fue descartado finalmente ya que su funcionamiento se basa en calidad de servicio y, por el tipo de aplicaciones empleadas en este trabajo, carecía de sentido. Se hace uso de aplicaciones *multithread* y establecer cuáles son los *threads* más prioritarios y cuáles los menos prioritarios no resulta coherente.

Capítulo 3

Diseño e implementación de algoritmos de reordenado de peticiones en Coyote

En este capítulo se describen las soluciones seleccionadas para implementar en el simulador. Para ello se detalla tanto el diseño de los diferentes algoritmos como su implementación en Coyote para ser capaz de reordenar peticiones en el controlador de memoria con independencia del tamaño y tipo del conjunto de datos de entrada o plataformas empleados.

Tras analizar los diferentes algoritmos del estado del arte, se ha considerado implementar los tres que han resultado más interesantes y adecuados para la realización de este proyecto. En primer lugar, se ha seleccionado el algoritmo *FRFCFS* ya que trata de maximizar el rendimiento a través de la localidad en el *Row Buffer*, adelantando las peticiones cuya fila objetivo coincida con la fila abierta actualmente. A continuación, se ha seleccionado el algoritmo *BLISS*, que no trata únicamente de obtener el mejor rendimiento, sino que toma en cuenta también otra métrica como es el *fairness*, todo ello sin incluir una complejidad excesiva en el planificador. Emplea un sistema de lista negra para ir clasificando los *threads* y que todos consigan ejecutar en un tiempo razonable. Y por último, se ha seleccionado el algoritmo *STFM* que realiza la planificación de peticiones de forma dinámica en función del *slowdown* de los diferentes *threads*. Proporciona calidad de servicio, tratando de mejorar el *fairness* gracias al equilibrio del desbalanceo en términos de ralentización para cada uno de los *threads*.

3.1. Análisis de requisitos

Para comenzar el capítulo, se va a realizar un análisis de los requisitos a tener en cuenta para la correcta realización del proyecto. No solo se debe tener en cuenta que su funcionamiento sea

el adecuado, sino que también se deben tener en cuenta ciertas métricas como el rendimiento o la eficiencia.

3.1.1. Requisitos funcionales

Los requisitos funcionales, por su definición, son aquellos que describen cualquier actividad que deba ser realizada por el sistema, es decir, la función que debe realizar el sistema cuando se cumplen una serie de condiciones. [21].

En el caso de este trabajo, el principal requisito funcional es la implementación de los diferentes algoritmos seleccionados en la herramienta Coyote. Para cada uno de ellos, se comprobará que la solución empleada es funcionalmente correcta, empleando diferentes métricas que podrán ser analizadas una vez finalizada la ejecución de la simulación a través de ficheros de log.

Además, el otro gran requisito funcional del trabajo es el análisis y comparación de los algoritmos implementados frente a las opciones disponibles previamente en el simulador. Se realizará un estudio de su rendimiento a través de diferentes métricas para comprobar cuál ha sido la ganancia obtenida.

3.1.2. Requisitos no funcionales

En cuanto a los requisitos no funcionales, se definen como atributos que pueden utilizarse para juzgar el comportamiento del sistema en lugar de su operación específica. [22]

Con esta definición en mente, los principales requisitos no funcionales a destacar en la realización de este proyecto son los siguientes:

- *Precisión*: La herramienta a modificar es un simulador, por lo que los resultados obtenidos deben ser lo más fieles posibles a los que se obtendrían al realizar la misma ejecución sobre una plataforma real.
- *Robustez*: El sistema debe ser capaz de operar independientemente de cual sea el conjunto de datos o plataforma empleados, mientras que estos se ciñan a las interfaces de la herramienta. En un sistema real se puede dar prácticamente cualquier combinación por lo que la herramienta debe estar preparada para trabajar con todas ellas. Además, para los mismos parámetros de entrada se deben obtener los mismos resultados, es decir, los resultados obtenidos no pueden variar en función de la ejecución que se realice.
- *Rendimiento*: El simulador debe ser capaz de realizar la ejecución en un tiempo asumible, ya que precisamente esa es su finalidad, obtener unos resultados lo más similares

posibles a la realidad, pero empleando un tiempo razonable, de forma que no resulte un obstáculo para el desarrollo de las actividades de diseño.

- *Extensibilidad y mantenibilidad*: Todas las modificaciones que se lleven a cabo en la herramienta deben ser compatibles con las capacidades ya existentes, de igual forma que las implementaciones deben realizarse con vistas a posibles mejoras en el futuro, así como dar la posibilidad de añadir nuevos algoritmos de reordenado de forma sencilla.

Si se cumplen todos los puntos mencionados en ambos apartados, nos aseguraremos de que el software desarrollado funcione de forma eficaz y sea capaz de dar una respuesta satisfactoria al problema planteado inicialmente.

3.2. Adaptación de los parámetros de entrada

El simulador Coyote soporta códigos RISC-V *baremetal* y cuenta con una serie de aplicaciones ya adaptadas a su ejecución en el simulador. Ejemplo de estas son *axpy*, encargada de realizar operaciones de álgebra lineal básica, *mt-matmul* que realiza multiplicaciones de matrices, o la que es objeto de estudio en este trabajo que es *spmv-vec*, que realiza operaciones de tipo matriz por vector.

En la figura 2 del anexo, se puede observar el algoritmo empleado para realizar el cálculo de la solución. Se realiza una división por filas en bloques y cada uno de ellos es asignado a un thread, que realizará la parte del trabajo correspondiente. Esta paralelización, como veremos más adelante, la modificaremos cambiando la granularidad para analizar cómo afecta en los patrones de acceso a memoria con los distintos algoritmos.

Las matrices con las que vamos a trabajar son las descritas en el paper de Luca Benini [8], que a su vez son extraídas de la web *SuiteSparse Matrix Collection* [23]. Estas tienen la particularidad de estar compuestas por datos de tipo *sparse*, lo que significa que los datos que predominan son ceros y los datos no nulos están muy separados entre sí. Estas se corresponden principalmente a problemas relacionados con la dinámica de fluidos y aplicaciones de ingeniería. En las siguientes figuras se puede observar una representación gráfica de la estructura de algunas de las matrices empleadas 3.1. Las zonas de color se corresponden con los datos, mientras que las zonas blancas, que en el caso de este tipo de matrices cubre la mayor parte del espacio, se corresponden con valores nulos.

Este tipo de matrices no se almacenan completas en un fichero ya que la mayoría de los datos no tienen valor, sino que se emplea un formato comprimido para reducir el tamaño de los ficheros. Las matrices quedan definidas a través de tres arrays: en el primero de ellos, cada elemento contiene el valor de la celda de la matriz (siempre que sea distinto de cero), el segundo indica la fila en la que se encuentra el elemento y el tercero el desplazamiento dentro de la fila. Con esta notación queda definida una matriz *sparse* al completo.

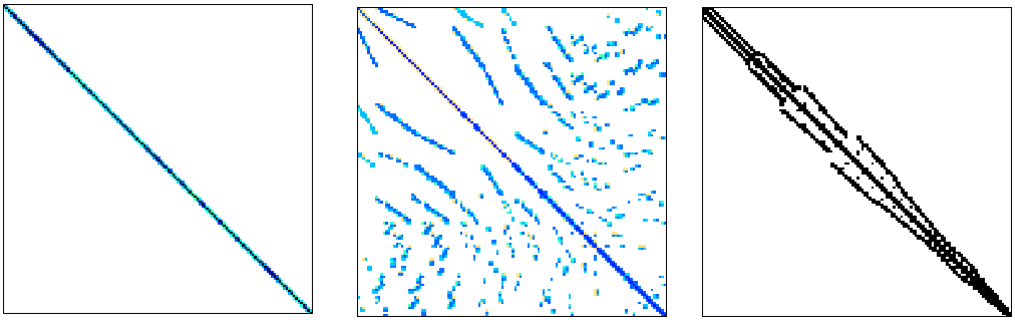

Figura 3.1: Matrices BenElechi1, msc01440 y nasa4704

Sin embargo, la aplicación "spmv-vec"no acepta como parámetro de entrada un fichero con esta definición, ya que, como se ha comentado, es un sistema *baremetal* por lo que no dispone de sistema operativo ni de las llamadas a sistema necesarias para facilitar la lectura de ficheros. Es por ello que la matriz debe tener un formato de fichero de cabeceras de C y por tanto, debe realizarse la transformación de los datos.

Para ello, se desarrolla un script en *Python* que toma como parámetro de entrada la matriz en formato comprimido y la transforma en el correspondiente fichero de cabeceras de C. El código realizado se puede ver en la figura 1 del anexo.

Una vez realizamos la transformación de todas las matrices, ya podemos realizar la compilación del programa con los parámetros adecuados para cada una de las ejecuciones.

3.3. Algoritmo de planificación FRFCFS

El algoritmo de planificación de memoria *FRFCFS* (First Ready First Come First Serve) busca maximizar el rendimiento del sistema minimizando la cantidad de cambios de fila en el *Row Buffer* [9]. Para lograrlo, da prioridad a las peticiones que resultan en un acierto de fila (es decir, aquellas que acceden a la fila actualmente cargada en el *Row buffer*) sobre aquellas que requieren cargar una nueva fila. Esto se debe a que cambiar de fila en el *Row Buffer* es una de las operaciones más costosas dentro del sistema de memoria: mientras que la selección y transferencia de datos al bus puede tardar entre 2 y 5 ns, un cambio de fila puede demorar alrededor de 40 ns, dependiendo de la tecnología utilizada. Esta diferencia de tiempo representa un problema importante, ya que, en el peor de los casos, cada nueva petición podría implicar un cambio de fila.

Este algoritmo se basa en otro más sencillo y conocido como es *FIFO*, en el que se van procesando las peticiones en el mismo orden en que van llegando. El gran beneficio de *FIFO* es su sencillez, ya que resulta muy fácil de implementar, pero precisamente esto es lo que hace que sea extremadamente ineficiente al emplearlo en entornos reales.

Con el algoritmo *FIFO*, cuando la petición que está en la cabeza de la cola se bloquea debido a falta de recursos, toda la cola se ve bloqueada, demorando así la ejecución del resto de peticiones y haciendo que el tiempo necesario para finalizar la tarea global sea mayor [9].

El algoritmo *FRFCFS* busca maximizar los aciertos en el *Row buffer* priorizando las solicitudes cuya fila objetivo coincide con la que se encuentra actualmente abierta. Estas peticiones se adelantan continuamente en la cola, ya que están listas para ejecutarse. En caso de que ninguna solicitud coincida con la fila activa en el *Row Buffer*, se procede al cambio de fila, y entonces prevalece el orden de llegada para atender las peticiones.

Este adelantamiento permite aprovechar la localidad de los datos en el *Row Buffer*, lo que lleva implícito una serie de beneficios como son la reducción de la latencia y el incremento del ancho de banda debido a que los tiempos de atención a las peticiones son menores y el sistema es capaz de procesar más en un menor tiempo.

Además, este algoritmo puede emplearse junto con otras estrategias para tratar de obtener todavía un mayor rendimiento. Ejemplo de ellas son la precarga de bancos o la optimización de filas, para conseguir el mayor número de aciertos en el *Row Buffer* antes de tener que hacer un cambio de fila. [9]

3.3.1. Ejemplo práctico

Supongamos que tenemos un controlador de memoria con las siguientes peticiones, mostradas de acuerdo a su orden de llegada:

- Solicitud A: Accede a la fila 1
- Solicitud B: Accede a la fila 2
- Solicitud C: Accede a la fila 1
- Solicitud D: Accede a la fila 3

Si se empleara el algoritmo *FIFO*, el orden que seguirían las peticiones sería: $A \rightarrow B \rightarrow C \rightarrow D$

Sin embargo, con el algoritmo *FRFCFS*, se priorizarían las peticiones que acceden a filas abiertas, resultando de la siguiente manera:

- A abre la fila 1
- C, que también requiera la fila 1, se procesa antes que B

- B abre la fila 2
- Y por último D abre la fila 3

Por lo tanto, el orden de peticiones con *FRFCFS* sería: $A \rightarrow C \rightarrow B \rightarrow D$

En la siguiente figura 3.2 se puede observar cuál sería el orden de procesamiento de las peticiones empleando ambos algoritmos:

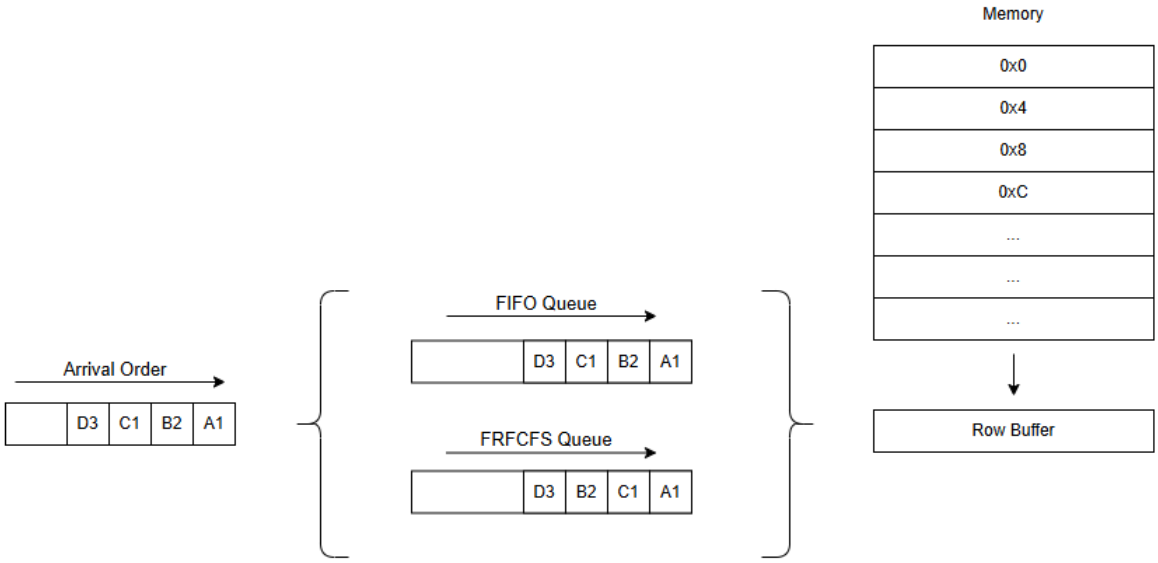

Figura 3.2: Ejemplo FIFO vs FRFCFS

Si atendemos a los parámetros temporales que se mencionan en la sección 2.2, podemos tener ligados a cada uno de los comandos de memoria los tiempos mencionados en la siguiente lista:

- *tPRE*: 8 ns
- *tCCD*: 4 ns
- *tRCD*: 7 ns
- *tCAS*: 8 ns
- *tRAS*: 18 ns
- *tBURST*: 8 ns

Empleando estos tiempos, se puede determinar el modelo de coste que obtendríamos con uno y otro algoritmo. En la realidad, el número de tiempos a considerar en el sistema de

memoria es mucho mayor, ya que hay diferentes casuísticas que deben ser tenidas en cuenta; pero, para simplificar los cálculos, se ha seleccionado un conjunto reducido de parámetros temporales.

Para el caso del algoritmo *FIFO*, las peticiones llegarían en los instantes 1, 2, 3 y 4. Como se desconoce el contenido del *Row Buffer*, lo primero que debe hacerse es una precarga asociada a la petición A (tPRE). Transcurrido ese tiempo se puede hacer la activación asociada a esa petición (tRCD) y su posterior lectura (tCAS). Finalizado ese tCAS los datos son transferidos al bus durante tBURST. Este patrón es seguido por todas las peticiones, pero hay una particularidad y es que el sistema de memoria funciona en *pipeline*, es decir, algunas fases de unas peticiones se solapan con otras fases de otras peticiones. En este caso, mientras se está haciendo la transferencia de datos al bus para la petición A, se puede comenzar con la precarga de la petición B, por lo que los tiempos asociados se solapan y el rendimiento aumenta. Si se realiza el desarrollo temporal completo, el resultado obtenido para este algoritmo es de 109 ns.

Sin embargo, al emplear el algoritmo *FRFCFS*, lo que ocurre es que la petición C se selecciona antes que la B, lo que supone que la fila objetivo sea la que ya esté en el *Row Buffer* y, por tanto, no hay que esperar los tiempos asociados a la precarga y activación de la fila, aunque sí que hay que tener en cuenta otros tiempos como el tCCD asociado al *delay* para poder seleccionar diferentes columnas de la misma fila. Gracias a esto se consigue mejorar la ocupación del bus de datos, aumentando el ancho de banda y, por consecuencia, el rendimiento. Nuevamente, si se realiza el diagrama temporal completo, el resultado obtenido para este algoritmo es de 89 ns, inferior a los 109 ns obtenidos con el algoritmo *FIFO*.

3.3.2. Implementación

El código fuente del simulador se localiza en el directorio *src* del proyecto alojado en el repositorio de Gitlab del BSC [24]. Prácticamente la totalidad de los elementos que componen la herramienta se encuentran en esta carpeta raíz, a excepción de la red de interconexión y el modelado de la memoria, que es la parte que nos interesa, por lo que la mayoría de las modificaciones se realizarán sobre este subdirectorio denominado *MemoryTile*.

Para implementar el algoritmo *FRFCFS* es necesario añadir dos nuevos ficheros en el directorio mencionado. Por un lado, se creará el *FRFCFSMemoryAccessScheduler.hpp* que contendrá la definición de los atributos y los métodos necesarios y, por otro lado, se creará el *FRFCFSMemoryAccessScheduler.cpp* que implementará todas las funciones mencionadas en el anterior fichero. La clase *FRFCFSMemoryAccessScheduler* hereda de la clase *MemoryAccessSchedulerIF* que es la clase padre de la que heredan todos los planificadores incluidos en el simulador. Esta contiene los atributos y métodos comunes para todos los planificadores y en cada uno de ellos únicamente deben incluirse aquellos que sean propios del algoritmo implementado.

En este caso, se añadirán 3 nuevos atributos en la clase para poder realizar la implementación

del algoritmo de forma correcta. En primer lugar se añade una lista de enteros *active_row_per_bank*, cuya longitud coincide con el número de bancos de la plataforma modelada y que almacena el número de fila activo en el *Row Buffer* para cada uno de los bancos. En segundo lugar se añade otra lista de enteros *request_position* que se encarga de almacenar la posición que ocupa la petición seleccionada para su ejecución en la cola de peticiones. Esta se utiliza para saber qué petición se debe completar cuando finaliza la simulación de la tarea. Y por último, se añade el atributo *num_bypass* empleado para monitorizar el funcionamiento del algoritmo. Este contador aumenta su valor cada vez que se produce un adelantamiento y nos permite conocer, una vez finalizada la simulación, el impacto que ha tenido al algoritmo en el rendimiento.

Los pasos fundamentales a seguir para obtener la siguiente petición a ejecutar, a modo de pseudocódigo, son los siguientes:

1. Para un banco determinado, que se recibe como parámetro, se recorre su cola de peticiones.
2. Si la petición de la iteración tiene como fila objetivo la fila abierta en el *Row Buffer* para ese banco, esta es seleccionada.
3. Si se llega al final de la cola y ninguna petición cumple la condición necesaria, la petición seleccionada para ejecutar es la cabeza de la cola.

Las peticiones recibidas son introducidas al final de la cola de peticiones del banco correspondiente.

También se debe modificar la clase *MemoryController.cpp*, añadiendo la opción del nuevo algoritmo implementado en función de lo seleccionado en el yaml de definición de la plataforma. El algoritmo a utilizar viene dado por el atributo *request_reordering_policy*. La clase *MemoryController* cuenta con un atributo denominado *sched* que es un puntero de tipo *MemoryAccessSchedulerIF* y, gracias al polimorfismo, es posible instanciar el planificador adecuado en función del valor indicado en el yaml, ya que todos los planificadores heredan de *MemoryAccessSchedulerIF*.

3.4. Algoritmo de planificación BLISS

Se puede observar que *FRFCFS* propone una mejora sobre el algoritmo básico *FIFO*, pero en cuanto se profundiza un poco, se puede comprobar que este tampoco resulta ser el algoritmo más eficiente.

Uno de los mayores problemas que se pueden dar al emplear el algoritmo *FRFCFS* es la inanición [9]. Esta se produce debido a que una petición que se encontraba cerca de la

cabeza de la cola puede verse adelantada por un número indeterminado de peticiones que fueron encoladas después, siendo el peor de los casos que nunca llegue a procesarse porque siempre haya peticiones más prioritarias (caso muy extremo pero teóricamente posible). Esto puede hacer que la aplicación vea muy lastrado su rendimiento y, lo que a priori parecía una mejora, se termine convirtiendo en una desventaja.

En este contexto surge *BLISS*, un algoritmo de planificación que trata de buscar un equilibrio entre rendimiento y fairness [10]. Permite planificar únicamente un número determinado de peticiones seguidas del mismo tipo, haciendo que el controlador de memoria no sea monopolizado por una única fila de memoria. En el momento en que se supera ese umbral, la petición a planificar será la siguiente que se corresponda con el orden de llegada. Esto nos permite reducir el número de veces que se realiza la apertura y cierre de línea en el *Row Buffer*, pero sin perjudicar en exceso a las peticiones de la cola que no atacaban contra esa línea de memoria.

Su funcionamiento se basa en crear un mecanismo de lista negra, en el que se irán añadiendo los *threads* en función del número de accesos seguidos realizados. Se dispone de un contador que se va actualizando con cada una de las peticiones planificadas, pudiendo producirse 2 escenarios:

- Si la petición a planificar y la que se ha planificado en última instancia, tienen como objetivo la misma línea de memoria, se incrementa el contador.
- Si la petición a planificar y la última planificada, atacan sobre diferentes líneas de memoria, el contador se resetea a 0

En el caso de que el contador supere un cierto umbral (configurable en el planificador), la petición se marcará como bloqueada, impidiéndose su planificación durante un número de ciclos determinado (también configurable). Cuando pasen esos ciclos, la petición se marcará como desbloqueada y esta será planificable de nuevo.

El orden de planificación de las peticiones es el siguiente:

1. Peticiones no bloqueadas
2. Acierto en el *row buffer*
3. Orden de llegada

Una vez que se ha computado la lógica de la lista negra, a la hora de seleccionar la siguiente petición a planificar, se comienza comprobando que la petición no se ha marcado como bloqueada, lo cual ya nos evita el problema de inanición. El siguiente paso es comprobar la línea objetivo de la petición, permitiendo el adelantamiento de peticiones en la cola en

caso de que esta coincida con la línea abierta en el *row buffer*. Y por último, si ninguna de las peticiones cumple con los anteriores requisitos, el planificador funcionará como una cola simple.

3.4.1. Ejemplo práctico

Supongamos que tenemos un sistema con 2 aplicaciones ejecutándose.

- Aplicación A: Solicitudes frecuentes y consecutivas a la misma fila de memoria
- Aplicación B: Solicitudes menos frecuentes pero también consecutivas

En un escenario en el que se utilice el planificador *FRFCFS*, la aplicación A tendería a monopolizar el sistema ya que hace un uso intensivo de las mismas filas de memoria, mientras que las peticiones de la aplicación B serían tratadas en último lugar, viéndose adelantadas por las de la otra aplicación.

Sin embargo, al utilizar *BLISS* esto no ocurre, ya que las peticiones de la primera aplicación estarían marcándose de forma recurrente como bloqueadas, permitiendo la ejecución de peticiones no tan regulares e intensas como las de la aplicación B.

Si el contenido de la cola de peticiones fuera el mencionado a continuación:
[A1, A2, A3, B1, A4]
Y los parámetros del planificador fueran un umbral de bloqueo de 2 y 20 ciclos de limpieza, lo que ocurriría es lo siguiente:

- Partiendo del estado base, se planifican las peticiones A1 y A2.
- Se alcanza el umbral por lo que la aplicación A se bloquea y permite la planificación de otro tipo de peticiones, reseteando el contador.
- Se planifica B1
- Pasados los 20 ciclos mencionados de desbloqueo, A se marca como planificable de nuevo y se seleccionan A3 y A4.

Como podemos observar, este planificador permite que la aplicación B haya podido ejecutar en un tiempo razonable, mientras que si se hubiera empleado *FRFCFS*, habría quedado relegada a la última posición, viéndose adelantada completamente por A.

En la figura 3.3 se puede observar cuál sería el orden de procesamiento de las peticiones empleando ambos algoritmos:

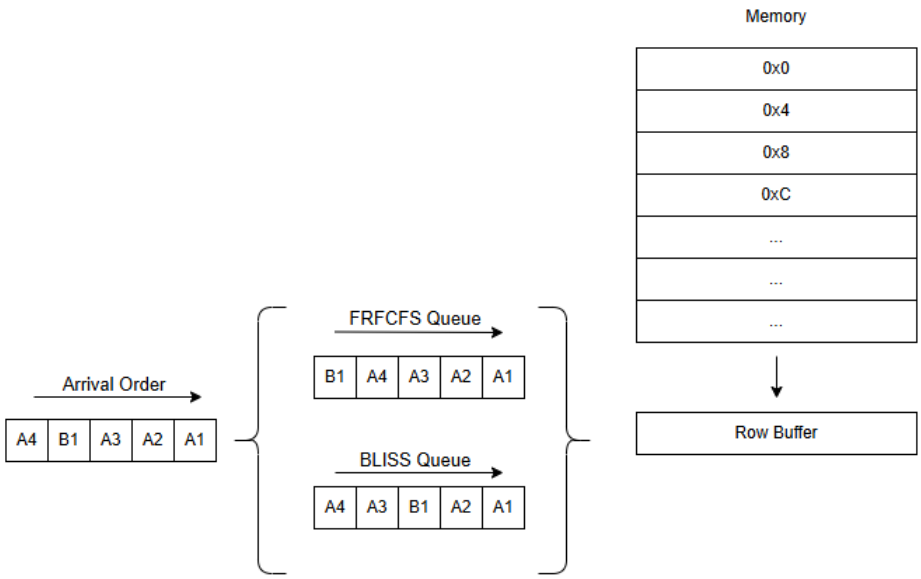

Figura 3.3: Ejemplo FRFCFS vs BLISS

En este caso, también se puede diseñar un modelo de coste con el que tener una estimación de la posible ganancia obtenida por el algoritmo *BLISS*.

Empleando los mismos tiempos que en el ejemplo anterior:

- $tPRE$: 8 ns
- $tCCD$: 4 ns
- $tRCD$: 7 ns
- $tCAS$: 8 ns
- $tRAS$: 18 ns
- $tBURST$: 8 ns

Las peticiones indicadas llegarían en los instantes 1, 2, 3, 4 y 5 respectivamente.

En el caso del algoritmo *FRFCFS*, en el primer instante se desconoce el contenido del *Row Buffer*, por lo que es necesario realizar una precarga asociada a la petición A1 (tPRE). Transcurrido ese tiempo se puede hacer la activación asociada a esa petición (tRCD) y su posterior lectura (tCAS). Una vez finalizado ese tCAS, los datos son transferidos al bus durante tBURST. Las posteriores peticiones (A2 y A3 por orden natural y A4 gracias al adelantamiento) tienen como fila objetivo la misma que está actualmente cargada en el *Row Buffer*, por lo que no es necesario realizar una nueva precarga y activación de fila, sino

que hay que tener en cuenta otro tiempo diferente asociado al *delay* para poder seleccionar diferentes columnas de la misma fila (tCCD). Una vez procesadas todas las peticiones de A, se pasaría a procesar B1, con su precarga y activación asociada. Si se realiza el esquema temporal completo y se tiene en cuenta el *pipeline* de determinadas fases, se obtendría un tiempo de unos 79 ns.

Sin embargo, al emplear el algoritmo *BLISS*, lo que ocurre es que no todas las peticiones de A pueden ejecutarse de forma consecutiva, sino que al planificar 2 peticiones, la aplicación queda bloqueada permitiendo la ejecución de la aplicación B. Para la primera petición A1, transcurren los tiempos tPRE, tRCD, tCAS y tBURST, mientras que para A2 simplemente es necesario esperar un tCAS adicional. En este punto A queda bloqueado y B puede comenzar a planificar sus peticiones. Se realiza la precarga (tPRE), activación (tRCD) y lectura (tCAS y tBURST) asociados a la petición B1. Transcurridos los ciclos de limpieza de la *blacklist*, los cuales se solapan con la ejecución de B, la aplicación A vuelve a ser planificable y, por tanto, se realizan la precarga, activación y lectura asociados a las peticiones A3 y A4. Nuevamente, si se realiza el desarrollo temporal completo, se obtendría un tiempo de unos 94 ns.

Como podemos observar, el tiempo empleado con el algoritmo *BLISS* es ligeramente superior respecto al algoritmo *FRFCFS*. Esto es normal, ya que el objetivo de *BLISS* no es maximizar el rendimiento, sino que tiene en cuenta el *fairness*. En el caso de este ejemplo, con el algoritmo *FRFCFS*, la petición B1 llega en el instante 3 pero no planifica hasta que todas las peticiones de A ya han sido ejecutadas, por lo que el desbalanceo en esta métrica es considerable. Sin embargo, con *BLISS*, esta es planificada una vez que se han ejecutado únicamente 2 peticiones de A. Si se realiza el cálculo del *fairness* relacionando el tiempo de llegada respecto al tiempo de atención, vemos cómo el algoritmo *BLISS*, a pesar de no conseguir el mejor rendimiento en todos los casos, obtiene un buen balanceo entre los *threads*.

Suponiendo una serie de 42 selecciones con ambos algoritmos sobre 4 programas, los resultados que obtendríamos serían, por ejemplo:

Para *FRFCFS*:

- P1: 20
- P2: 15
- P3: 5
- P4: 2

Y para *BLISS*:

- P1: 13
- P2: 11

- P3: 9
- P4: 9

Si se realiza el cálculo del *fairness*, se puede determinar que el valor resultante para *FRFCFS* es aproximadamente 0,674; mientras que para *BLISS* resultaría aproximadamente 0,976. Esto se debe a que con *FRFCFS* es posible priorizar mucho un programa respecto a otros, como en este caso ocurre con P1 y P2, mientras que con *BLISS*, la elección es más repartida ya que en el momento en que un programa se planifique constantemente es bloqueado.

Este algoritmo también presenta una serie de limitaciones, como por ejemplo, la falta de flexibilidad ante comportamientos dinámicos en las aplicaciones que cambian su patrón de acceso durante la ejecución, ya que *BLISS* clasifica las aplicaciones en función del número de accesos consecutivos que se realizan sobre la memoria. Otra limitación reside en la buena elección que se haga para el tiempo de reinicio de la *blacklist*, si es un valor muy alto, se penalizará demasiado a la aplicación bloqueada, mientras que si el valor es muy bajo, el impacto que tendrá el algoritmo será menor. Una buena elección para este parámetro determinará el rendimiento total obtenido con el algoritmo.

Además, también se debe tener en cuenta la dificultad de implementación para tratar de obtener el mayor equilibrio rendimiento - coste. Otras propuestas pueden conseguir a priori un mayor rendimiento en determinadas situaciones (como *STFM* que se detallará más adelante), pero pueden no resultar interesantes debido a la diferencia entre la ganancia de rendimiento y la dificultad de implementación.

3.4.2. Implementación

Para la implementación de este algoritmo, se deberán añadir 2 nuevos ficheros, al igual que en el caso anterior. Por un lado, se creará el *BLISSMemoryAccessScheduler.hpp* que contiene la definición de los atributos y los métodos necesarios y, por otro lado, se creará el *BLISSMemoryAccessScheduler.cpp* que realizará la implementación de todas las funciones definidas en el otro fichero.

En este caso, adicionalmente a los atributos añadidos en la implementación del algoritmo *FRFCFS* en la sección 3.3.2, se deberán añadir seis nuevos atributos. En primer lugar, se crea un mapa que actúa como *blacklist* en la que llevar la cuenta de los *threads* que han sido bloqueados por planificar un número excesivo de peticiones seguidas. Además, es necesario incluir cuatro variables: un contador de peticiones consecutivas, el identificador de la última aplicación cuya petición de memoria ha sido planificada, el umbral a partir del cual las aplicaciones son agregadas a la *blacklist*, y una variable que indica el periodo de refresco utilizado para sacar las aplicaciones bloqueadas de la *blacklist*.

Por último, a modo de monitorización, se incluye un mapa en el que se almacena el número de veces que se bloquea cada una de las aplicaciones. Esto nos permite tener un registro con

el que comprobar que el algoritmo está funcionando adecuadamente y está teniendo impacto en la ejecución de la simulación.

Las inserciones en la cola de peticiones, al igual que para el algoritmo *FRFCFS*, se siguen haciendo al final de la cola.

Para la obtención de la petición a planificar, se emplea el algoritmo descrito en el siguiente pseudocódigo:

1. Se obtiene la cabeza de la cola de peticiones
2. Se comprueba si la aplicación asociada coincide con la última aplicación planificada. En caso de coincidir se aumenta el contador de peticiones consecutivas y, sino, se resetea el contador y la última aplicación planificada es la actual.
3. Se comprueba si el contador de peticiones consecutivas supera el umbral y, en caso de cumplirse, la aplicación es agregada en la blacklist
4. Se comprueba si la aplicación de la petición seleccionada está en la blacklist. Si no está, esta es la petición planificada. Si está, se recorre la cola de peticiones en busca de la primera petición que no esté bloqueada. Si ninguna cumple esa condición, se emplea FRFCFS, tratando de planificar una petición cuya fila objetivo coincida con la que está abierta en el Row Buffer. Si ninguna petición cumple tampoco esa condición, se planifica la seleccionada originalmente

Además, en la clase *MemoryController.cpp*, la cual lleva la cuenta de los ciclos simulados, se deberá comprobar que, cada vez que se cumpla el intervalo de reseteo de la *blacklist*, se haga uso del método *clearBlacklist*, permitiendo que todas las aplicaciones vuelvan a estar disponibles para ser planificadas.

Para la monitorización de este algoritmo, se han añadido una serie de estadísticas a través de *Sparta* para indicar las veces que ha sido agregada a la *blacklist* cada una de las aplicaciones. Para ello, se modifica el destructor de la clase en la que se implementa el planificador, haciendo que ejecute el método *showData* y mostrando el contenido de las estadísticas recopiladas con el mapa *num_activaciones*.

3.5. Algoritmo de planificación STFM

El algoritmo STFM (Stall-Time Fair Memory Scheduler) es una propuesta para gestionar el acceso equitativo a la memoria DRAM en chips multiprocesador (CMP) [11]. Este tipo de sistemas obtienen un rendimiento superior frente a los sistemas tradicionales, pero acarrean otro tipo de problemas diferentes ya que suelen compartir recursos hardware, como la

memoria, entre varios hilos de procesamiento. Sin una gestión adecuada, los hilos pueden experimentar interferencias entre ellos, lo que lleva a desbalanceos en el rendimiento y a la incapacidad de garantizar una calidad de servicio uniforme.

Estos efectos adversos pueden ir desde pérdidas de rendimiento hasta brechas de seguridad [11]. El primero de los casos genera en el usuario sensación de incertidumbre, ya que se disminuye la predictibilidad de la ejecución de las aplicaciones porque otras están interfiriendo, haciendo un uso indebido de los recursos del sistema. Esto no es lo deseable, pero no resulta tan grave como que *threads* de otras aplicaciones malintencionadas monopolicen todos los recursos del sistema y, por tanto, se produzca una denegación de servicio.

En sistemas CMP, tradicionalmente se han empleado algoritmos que optimizan el ancho de banda total de la memoria, como *FRFCFS* que trata de priorizar las solicitudes aprovechando las filas abiertas en memoria, pero no considera el impacto en el rendimiento de los *threads* individuales.

El algoritmo STFM redefine la justicia en el acceso a la memoria DRAM. En lugar de asignar equitativamente el ancho de banda, el objetivo es igualar la ralentización relacionada con la memoria que experimenta cada hilo. Esto se mide mediante dos variables fundamentales que son almacenadas por parte del planificador para cada thread:

- *Tshared,* T_s: Tiempo de espera experimentado por parte del thread mientras comparte la DRAM con otro hilos. Para calcular este parámetro, se emplea un contador que incrementa su valor cada vez que no se puede realizar el *commit* de instrucciones debido a un fallo en la cache L2.
- *Talone,* T_a: Tiempo de espera que experimentaría el thread si ejecutara en solitario. En el cálculo de este parámetro se tienen en cuenta diferentes aspectos como la interferencia del bus DRAM o la interferencia en los bancos de memoria.

Estos dos valores se relacionan entre sí dando lugar a una métrica conocida como "Memory Slowdown" $\rightarrow S = \frac{T_s}{T_a}$. El algoritmo *STFM* tiene como objetivo minimizar la diferencia entre los valores S de todos los hilos, garantizando un acceso equitativo. De esta fórmula se deduce que si el thread se ve afectado por muchas interferencias de otros hilos, su valor de S será alto, mientras que si ejecuta solo la mayor parte del tiempo, el valor de S será más pequeño.

El planificador priorizará los threads basándose en su "Memory Slowdown", seleccionando aquellos cuyo valor sea más alto y, de esta manera, la ralentización se repartirá entre todos los threads.

El funcionamiento de *STFM* viene dado por las siguientes reglas:

1. *Cálculo del desbalanceo*: Se determina el desbalanceo en la ralentización entre el thread con mayor ralentización (Smax) y el de menor ralentización (Smin)

2. *Evaluación del sistema*: Si $Smax/Smin <= \alpha$, donde α es un umbral configurable, se priorizan las particiones usando la política *FRFCFS*. De lo contrario, se aplica una regla de justicia que da prioridad a los hilos con mayor ralentización.

3. *Priorización*: Dentro de los threads priorizados, se siguen reglas similares a *FRFCFS*, atendiendo primero las solicitudes que son aciertos en el *Row Buffer* y luego las más antiguas.

De los dos valores necesarios para el cálculo del "Memory Slowdown", el *Tshared* es, en principio, sencillo de determinar, ya que un hilo habitualmente comparte recursos con el resto de threads en el sistema. Sin embargo, el cálculo del *Talone* es más complejo, ya que depende del comportamiento hipotético del hilo al ejecutarse en solitario. El planificador *STFM* estima este valor basándose en el tiempo extra de espera causado por la interferencia de otros hilos *Tinterference*.

Este valor *Tinterference* viene dado tanto por interferencias en el bus, debido a conflictos en el uso del bus de datos, como por interferencias en los bancos de memoria, generadas por solicitudes en paralelo de los mismos bancos de memoria.

En cuanto a la implementación, *STFM* requiere que existan registros por thread para almacenar los valores de *Tshared*, *Tinterference* y el valor de slowdown *S*. Además, debe incluirse en el controlador de memoria la lógica necesaria para ir calculando las prioridades en función del valor de slowdown, y se debe configurar el valor de α adecuándolo a las necesidades del sistema operativo o la carga del sistema.

El parámetro α es un valor configurable en el planificador que determina el umbral a partir del cual se considera que un *thread* está viéndose excesivamente ralentizado respecto a los demás. Si el ratio entre el *max slowdown* y el *min slowdown* es inferior al valor de α, el algoritmo empleado en la planificación será *FRFCFS*, mientras que si el ratio es superior al valor de α, se empleará *STFM*, seleccionando el *thread* con mayor *slowdown*.

Entre los beneficios de este planificador, se puede destacar el fairness mejorado entre *threads*, ya que se reduce significativamente la variación en la ralentización entre ellos; su mayor rendimiento global, aumentando el throughput del sistema al evitar el bloqueo de hilos con pocas solicitudes; y su escalabilidad, ya que su desempeño mejora en sistemas con mayor cantidad de núcleos, manteniendo un balance entre justicia y rendimiento.

3.5.1. Ejemplo práctico

Supongamos un procesador con 4 cores que ejecuta simultáneamente 4 *threads*:

- *Thread* A: Accede a memoria de forma intensiva con alta localidad en el *Row Buffer*

- *Thread* B: Realiza accesos menos intensivos y además tiene baja localidad en el *Row Buffer*
- *Thread* C: Tiene accesos a memoria distribuidos uniformemente entre los bancos de DRAM
- *Thread* D: Emite solicitudes a memoria de forma intermitente (en ráfagas)

Si se utilizara un planificador como *FR-FCFS*, el resultado que se obtendría es que el *thread* A, debido a su alta localidad en el buffer de filas, sería priorizado constantemente. Los *threads* B y C sufrirían penalizaciones porque sus accesos requieren cambios de fila, que tienen una mayor latencia. Y, por último, el *thread* D podría experimentar largos tiempos de espera debido a que emite solicitudes menos frecuentemente.

Esto provocaría un desequilibrio en el rendimiento, donde el *thread* A tendría prioridad constantemente, mientras que los otros *threads* pueden quedar severamente ralentizados.

Sin embargo, al emplear *STFM* se busca igualar la ralentización experimentada por cada *thread* debido a las interferencias en la memoria compartida.

Al inicio, asumiendo por ejemplo los siguientes tiempos, el "Memory Slowdown" calculado para cada uno de los hilos es el siguiente:

- Programa A: Tshared/Talone = 1.2
- Programa B: Tshared/Talone = 3.0
- Programa C: Tshared/Talone = 2.5
- Programa D: Tshared/Talone = 1.8

El programa B es el que tiene un mayor valor de (Smax), por lo que sus peticiones se verán priorizadas hasta que la ralentización se equilibre. En el momento en que la diferencia entre Smax y Smin sea inferior a α (por ejemplo, 0.5), entonces *STFM* dejará de priorizar en función del *Memory Slowdown* y se comportará como *FR-FCFS* favoreciendo los aciertos en el *Row Buffer*.

Este mecanismo garantiza que se reduzca la ralentización entre los programas y se mejore el rendimiento global del sistema, evitando que algunos programas sufran inanición.

3.5.2. Implementación

Este último algoritmo a implementar resulta también el más complejo, ya que requiere dotar de una mayor inteligencia al sistema para determinar el estado en el que se encuentran los diferentes componentes en cada paso de la simulación.

Al igual que para *FRFCFS* y *BLISS*, se deben crear dos nuevos ficheros, el *STFMScheduler.hpp* en el que se definen los atributos y métodos a utilizar, y el *STFMScheduler.cpp* en el que se realiza la implementación de esos métodos.

La inserción de las peticiones en la cola de peticiones se realiza al final de la cola, al igual que en el resto de algoritmos.

En cuanto a atributos, se debe crear un mapa por cada una de las variables implicadas en el algoritmo. Como se ha visto en la parte teórica, se debe realizar el seguimiento de los valores *Tshared*, *Talone*, *Tinterference* y *Slowdown*. Estos se irán actualizando en cada iteración con los valores correspondientes del simulador.

En primer lugar, se desarrolla el método *updateSlowdowns* que es utilizado de forma previa a la elección de la petición. Esto determinará cómo se realizará la planificación en función de si se supera el umbral o no. Se actualiza el valor de *Slowdown* para cada uno de los threads en base a los valores de *Tshared*, *Talone* y *Tinterference*.

Se desarrolla el método *updateTshared* que actualiza su valor cada vez que se produzca un fallo de cache en la L2. Para ello se implementarán una serie de métodos en las clases *MemoryController.cpp* y *ExecutionDrivenSimulationOrchestrator.cpp* para determinar cuándo se produce un fallo en la cache L2 durante la simulación. La clase *ExecutionDrivenSimulationOrchestrator.cpp* llevará la cuenta del número de ciclos perdidos por fallos producidos en la cache L2 para cada uno de los *threads* y su valor será leído por el planificador. El método *updateTshared* será utilizado cada vez que se planifique una nueva petición.

Se desarrolla también el método *updateInterference*, que será empleado cada vez que una petición realice su método *issue*, calculándose así su latencia. Este valor deberá ser mantenido tanto para el *thread* cuya petición es planificada, como para el resto de *threads*.

El valor de interferencia para el resto de *threads* se desglosa en dos apartados fundamentales:

- Interferencia debida al bus *DRAM*: Aquí debemos tener en cuenta el tipo de tecnología *DRAM* modelada, el *delay* asociado a la longitud del bus y la longitud de la ráfaga. El bus es un medio compartido por lo que únicamente una petición puede emplearlo en cada instante de tiempo.
- Interferencia debida a bancos *DRAM*: Esta es debida a que los demás *threads* pueden tener peticiones listas para ser planificadas al mismo banco que la petición seleccionada, y tienen que esperar hasta que esta sea atendida. Para realizar el cálculo, se recorrerá la cola de peticiones pendientes en ese banco y se actualizará el valor de interferencia para los *threads* que tengan alguna petición encolada.

El valor de interferencia para el propio *thread* también debe actualizarse debido al sistema de memoria *DRAM* compartida. Dos peticiones consecutivas a la misma fila del mismo banco

supondrían un acierto si ejecutara solo. Sin embargo, otros *threads* podrían modificar la información en el *Row Buffer* y provocar un fallo con su consiguiente penalización. Para realizar el cálculo se debe tener en cuenta el número de peticiones que tiene pendientes el *thread* seleccionado en el resto de bancos.

3.6. Otras mejoras planteadas en el simulador

Adicionalmente a las implementaciones de los diferentes algoritmos, se han realizado otra serie de mejoras que no están relacionadas implícitamente con el desarrollo de este proyecto pero que aportan valor al simulador. La mayoría de estas mejoras están enfocadas a la información que arroja la herramienta cuando finaliza la ejecución de la simulación.

En primer lugar, se añade una nueva métrica hasta ahora inexistente, que es el número de ciclos simulados por core. La herramienta ya mostraba cuál era el número de ciclos simulados en total al finalizar la ejecución de la aplicación, pero con este cambio, además se muestra el desglose de ciclos por core.

Por otro lado, se ha añadido otro conjunto de métricas que permiten determinar la saturación del sistema de memoria. Se analiza el estado de las colas de peticiones regularmente y, en función de su ocupación, se puede determinar si el elemento que lastra el rendimiento de la aplicación es la memoria.

Se añaden cuatro contadores, cada uno dedicado a un rango de peticiones: menos de 5, entre 5 y 10, entre 10 y 15 y más de 15. El valor deseable para no tener congestión en el sistema de memoria es tener cuantas menos peticiones en cola mejor, pero como veremos en la sección 4.2 esto no es siempre así. Esta información es añadida a la salida de los resultados cuando finaliza la simulación.

Además, se ha modificado el código de la aplicación empleada *spmv-vec* para dotar de una mayor granularidad a la paralelización realizada. Como se comentó en la sección 3.2, la aplicación original realiza una división de las filas por bloques y estos son asignados a los distintos *threads*. El cálculo de los bloques se realiza teniendo en cuenta únicamente el número de cores (un thread por core) y el número de filas, pero, en función de la organización de la información en la matriz, el rendimiento resultante podría verse afectado debido a la arquitectura de la memoria y la localidad de datos.

Se propone una modificación del algoritmo que permite realizar una división alternativa de las filas, haciendo que la composición de los bloques sea diferente. El número de filas que procesa cada uno de los *threads* es el mismo respecto a la versión anterior, pero la composición de los bloques cambia. El nuevo algoritmo propuesto se recoge en la figura 3 del anexo.

Capítulo 4

Evaluación

En este capítulo se detallan los diferentes experimentos que se han realizado en el simulador *Coyote* para, por un lado, comprobar el correcto funcionamiento de los algoritmos implementados descritos en el capítulo 3, y por otro lado, realizar una comparativa tanto entre ellos como frente a los algoritmos previamente existentes. Los experimentos se explican de tal forma que cualquier persona es capaz de replicarlos y obtener los mismos resultados.

4.1. Metodología

En primer lugar, se debe definir correctamente la forma en la que se van a realizar los experimentos para comprobar que la implementación de los algoritmos realizada es la correcta. Esto resulta esencial ya que, si no estamos completamente seguros de que el funcionamiento es el esperado, los resultados que obtendríamos al realizar los diferentes *benchmarks* no resultarían veraces.

Por ello, para cada uno de los algoritmos, se incluyen una serie de métricas que validan su comportamiento. En el caso del algoritmo *FRFCFS* se muestra en la salida resultante de la ejecución de la simulación, el número de adelantamientos de peticiones realizados en las colas. Para el algoritmo *BLISS* se muestra en la salida el número de veces que una aplicación fue enviada a la *blacklist* y, por tanto, dio pie a que otra aplicación menos demandante pudiera planificarse. Y, por último, para el algoritmo *STFM* se muestra en diferentes intervalos el *slowdown* del sistema para validar si realiza la planificación empleando *FRFCFS* cuando el valor es inferior al umbral, o selecciona el *thread* con mayor *slowdown*.

Las distintas pruebas realizadas tienen como objetivo determinar el rendimiento del sistema al emplear cada uno de los algoritmos implementados y realizar una comparativa tanto entre ellos como con los ya existentes en la herramienta. Se modificarán diferentes parámetros para tratar de simular el mayor número de escenarios posible y así emular el comportamiento de

una máquina real. Los experimentos se realizan con una complejidad creciente, partiendo de plataformas más sencillas, hasta llegar a escenarios con una alta demanda de recursos.

Para que los experimentos sean adecuados se debe seguir un orden a la hora de su diseño, además de tener en cuenta diferentes aspectos. Los pasos mostrados a continuación describen el proceso de definición de los experimentos:

- *Definir el objetivo del experimento*: En este primer apartado se deben decidir cuáles van a ser las metas del experimento. *Coyote* muestra en la salida de la ejecución múltiples métricas que podemos analizar para conocer el rendimiento del sistema. Pueden ser tanto las mencionadas anteriormente para determinar la correctitud de los algoritmos, como otras propias de la ejecución en si, como el tiempo de simulación, el número de accesos a memoria realizados o el uso de las caches. En función del objetivo que marquemos, se adaptarán los experimentos para conseguir profundizar en cada aspecto en una mayor o menor medida.

- *Diseñar la plataforma*: Una vez tenemos claro cuál es el objetivo de la prueba, se debe definir la plataforma sobre la que se van a realizar los experimentos. El simulador *Coyote* recibe como parámetro al lanzar la ejecución un fichero con extensión *YML* en el que se definen las valores que queremos emplear en la simulación. Ejemplos de estos parámetros son el *bypass* de cada una de las caches, la aplicación a ejecutar, el número de cores, la frecuencia, las características de las memorias o el algoritmo de planificación a utilizar, entre otros muchos. Para los experimentos realizados en este proyecto se ha empleado un fichero denominado *simple_arch.yml* que se va modificando a través de las diferentes ejecuciones para adaptarlo al experimento correspondiente.

- *Estimar los resultados*: Una vez tenemos todas las partes definidas y la simulación está lista para comenzar, debemos prever de forma teórica los resultados que esperamos obtener ya que si no, no seríamos capaces de sacar conclusiones a partir de los valores obtenidos. En este tipo de simulaciones tan extensas y complejas, es imposible determinar con exactitud cuales son los valores que vamos a obtener, pero si nos podemos hacer una idea de lo que va a ocurrir de forma general, en función del apartado que modifiquemos.

- *Realizar el experimento*: Se realiza la ejecución lanzando la simulación por terminal a través del comando "./coyote -c simple_arch.yml". Esto realiza una ejecución, pero como queremos analizar la variación de rendimiento del sistema a través de la modificación de diferentes parámetros, se han desarrollado una serie de *bash scripts* con los que automatizar los *benchmarks*. Estos se encargan de lanzar las diferentes simulaciones modificando los valores deseados del fichero de configuración de la plataforma *simple_arch.yml*

- *Análisis de los resultados*: Una vez finalizado el experimento, se comprueban los resultados obtenidos para verificar si el planificador se ha comportado de la forma adecuada o ha ocurrido algo inesperado. La salida habitual del simulador es a través de la consola, por lo que para poder analizar los resultados y trabajar con los valores retornados,

se redirige la salida a un fichero que es depositado en el directorio correspondiente. Se han desarrollado una serie de scripts en *Python* que se encargan de leer ese tipo de ficheros de salida y generar gráficas de las métricas deseadas.

Con todos estos datos ya tenemos las herramientas suficientes para valorar si el experimento ha resultado satisfactorio.

Una vez concluida la prueba, se puede determinar si ésta ha sido un éxito o un fracaso en función de los resultados obtenidos. Si los valores que devuelve el simulador se asemejan a los que teníamos previstos obtener teóricamente, podemos concluir que el algoritmo se comporta correctamente. En algunas ocasiones puede ocurrir que los resultados obtenidos no coincidan con los previstos y debemos parar a estudiar cuál ha sido el motivo de esa variación. Esta puede deberse a una anomalía en una ejecución o a un planteamiento erróneo, por lo que, gracias a la experimentación, somos capaces de corregirlo.

Al tratarse de un entorno simulado, no es necesario repetir los experimentos un número determinado de veces para obtener un resultado promedio, ya que los simuladores son deterministas y en todas las ejecuciones, para los mismos parámetros de entrada, se obtendrán los mismos resultados.

Todas las pruebas han sido realizadas en un ordenador portátil, concretamente el modelo MSI Modern 15 A11SB-011ES que cuenta con un procesador Intel Core i7-1165G7, 16 GB de memoria RAM, 1 TB de almacenamiento de estado sólido y una tarjeta gráfica Nvidia MX450, aunque esta no resulta relevante ya que no se emplea en la simulación. El sistema operativo empleado es Ubuntu en su versión 24.04, con el kernel de Linux versión 6.8.0-57-generic.

4.2. Experimentos

En esta sección se realiza la comprobación de la correcta implementación de los algoritmos descritos en el Capítulo 3 y una comparativa, en términos de rendimiento, de los 4 algoritmos que quedan implementados en el simulador. Los experimentos se detallan para que puedan ser replicados por cualquier usuario.

En primer lugar, es necesario definir la plataforma que se va a emplear a través del fichero *simple_arch.yml*. Los diferentes parámetros empleados quedan recogidos en la tabla B.1 del anexo. En ella se puede observar que la mayoría de los parámetros son fijos a excepción de 3, *vector_bypass_l1*, *num_cores* y *request_reordering_policy*. Estos se irán modificando a través de los diferentes experimentos para determinar cómo se comportan los algoritmos en distintas situaciones.

Para cada uno de los algoritmos se realizará una batería de pruebas empleando el set de 13 matrices disponibles y variando tanto el número de cores simulados (4, 8, 16, 32 y 64) como

el uso o no de la memoria cache L1.

Las características de las matrices empleadas quedan recogidas en la tabla 4.1:

Nombre	**Número de filas**	**Número de valores no nulos**
BenElechi1	245874	13150496
circuit5M_dc	3523317	14865409
pwtk	217918	11524432
Dubcova1	16129	253009
exdata_1	6001	2269500
fv1	9604	85264
G3_circuit	1585478	7660826
hood	220542	9895422
msc01440	1440	44998
Na5	5832	305630
nasa4704	4704	104756
s2rmq4m1	5489	263351
thermal2	1228045	8580313

Cuadro 4.1: Set de matrices empleados en los experimentos

La ejecución de las pruebas se realiza a través de diferentes *bash scripts* para poder automatizarlas y que de esta forma resulte más sencillo ejecutarlas y evitar posibles errores en las modificaciones de los ficheros de configuración.

Antes de comenzar con la comparativa del rendimiento obtenido por los diferentes algoritmos, debemos asegurar su correcta implementación. Para ello, como se mencionó anteriormente, se han introducido una serie de métricas en el simulador para garantizar que el comportamiento de los algoritmos implementados es el esperado. Para cada uno de ellos se generarán métricas personalizadas que permitirán conocer cómo ha sido la ejecución del algoritmo y, comparándolas con los resultados teóricos previamente analizados, asegurar su funcionamiento.

En el caso del algoritmo *FRFCFS*, la métrica que nos interesa conocer es el número de adelantamientos que se realizan durante la simulación. Gracias a este dato, podemos determinar que existen peticiones de memoria que están siendo adelantadas respecto a otras en función de si la fila objetivo coincide con la fila abierta en el *Row Buffer*.

Se realizan una serie de pruebas con el set de matrices y un número variable de cores, y el resultado promedio es el que se recoge en la imagen 4.1. Como se puede observar, el número de adelantamientos crece según va aumentando el número de cores, tanto para *bypass* de L1 como sin *bypass*. Este resultado es el esperable y se debe principalmente a que, al existir más

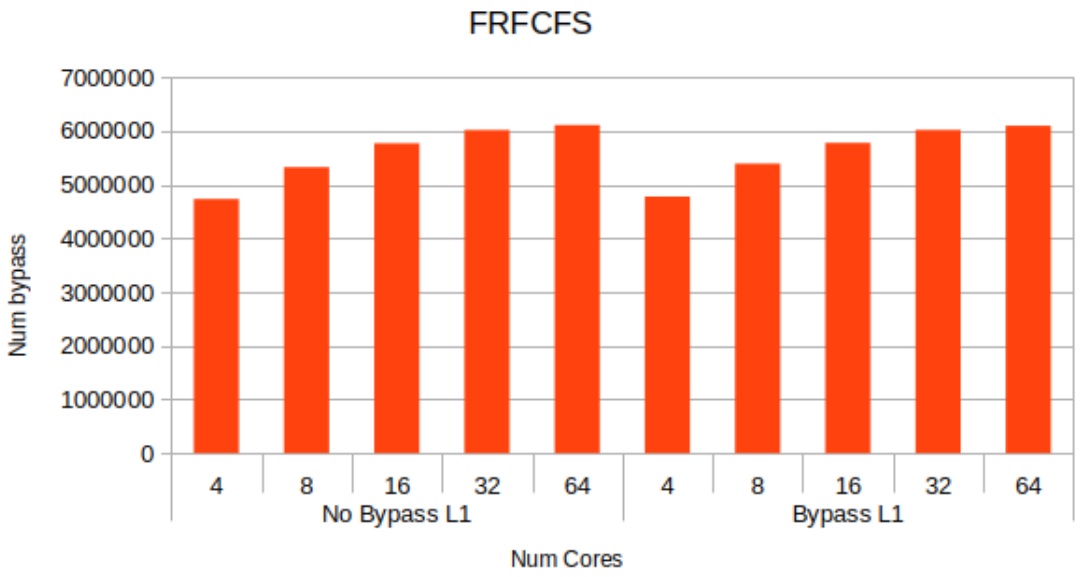

Figura 4.1: Adelantamientos realizados con FRFCFS

peticiones de memoria de diferentes *threads*, en cada iteración la probabilidad de que alguna de ellas tenga como fila de memoria objetivo la misma fila que está abierta en el *Row Buffer* es mayor y, por tanto, se pueden producir más adelantamientos.

Por otro lado, para el algoritmo *BLISS*, la métrica que realmente nos interesa conocer para comprobar si el algoritmo está funcionando como debería, está relacionada con el número de veces que un *thread* es incluido en la lista negra. Gracias a esto, conseguimos conocer cómo se han ido priorizando los diferentes *threads* en función de los bloqueos gestionados por el planificador.

Analizando de forma previa los posibles resultados, lo esperable es que el número de veces que algún *thread* es incluido en la *blacklist* disminuya a medida que se va aumentando el número de cores. Esto se debe a que al existir más posibilidades de elección a la hora de realizar la selección por parte del planificador, es menos probable que un mismo *thread* se elija múltiples veces consecutivas.

Como se puede observar en la imagen 4.2, las hipótesis realizadas previamente son ciertas y el número de veces que algún *thread* es enviado a la *blacklist* disminuye al aumentar el número de cores. Para niveles altos de cores, el número de bloqueos es de apenas unas decenas, tanto con *bypass* de L1 como sin *bypass*, ya que es muy complicado que peticiones de un mismo *thread* sean seleccionadas durante muchos ciclos seguidos. Sin embargo, con un número bajo de cores, las posibilidades de elección son inferiores, por lo que se producen más bloqueos.

Por último, para el caso del algoritmo *STFM*, la comprobación que se realiza de su correctitud, es ir comprobando, a medida que se va realizando la simulación, cómo evoluciona el valor de "Memory Slowdown". Cada ciertos intervalos de tiempo se muestra el valor de la

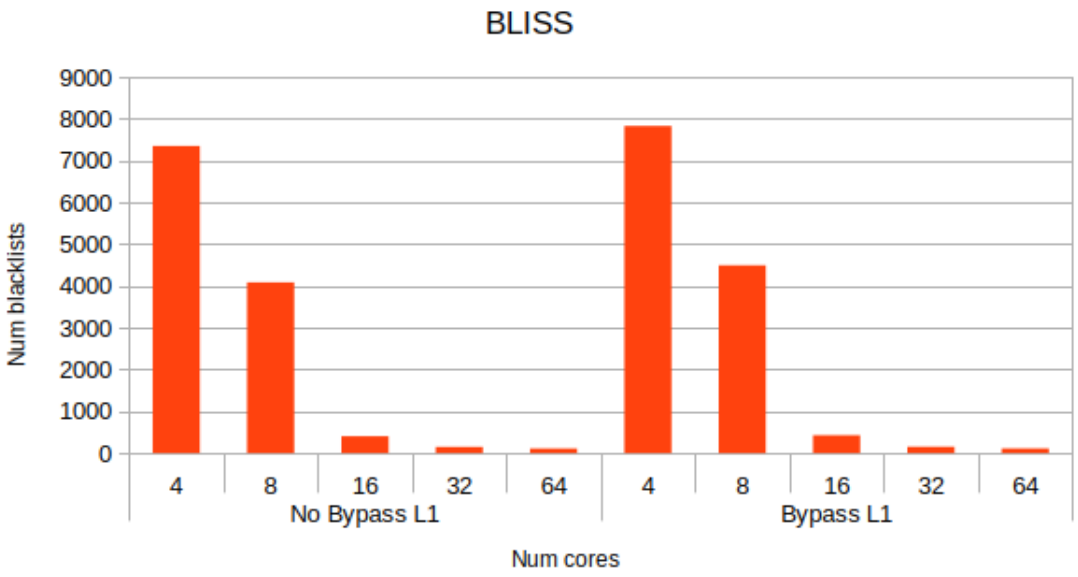

Figura 4.2: Threads incluidos en la *blacklist* con BLISS

métrica por pantalla y, de esta forma se puede comprobar, en base al umbral establecido, cómo va variando el algoritmo entre el propio *STFM*, cuando hay mucho desbalanceo entre *threads* y *FRFCFS* cuando todos los hilos tienen una ralentización similar.

Una vez comprobado el funcionamiento de los diferentes algoritmos, las siguientes pruebas nos permitirán determinar cuál es el rendimiento de las diferentes implementaciones realizadas.

Tras la realización de las pruebas, se obtienen una gran cantidad de datos que son procesados y analizados para determinar los resultados de la implementación de los diferentes algoritmos en el simulador.

La aproximación seguida a la hora de obtener conclusiones a partir de los datos es de forma jerárquica, partiendo de los resultados generales y deteniéndose en los casos que sean dignos de mención, ya que, como se ha mencionado antes, el volumen de resultados es tan grande que no se puede analizar cada caso de forma individual.

El rendimiento de los diferentes algoritmos implementados será analizado de forma relativa respecto al algoritmo *FIFO* que será empleado como *baseline*.

En primer lugar, la gráfica 4.3 muestra la ganancia media obtenida en función del número de cores para cada uno de los diferentes algoritmos desarrollados, sin hacer uso del *bypass* de la cache L1. Como se puede observar, en el caso del algoritmo *FRFCFS* este rendimiento va en aumento a medida que se va aumentando el número de cores hasta alcanzar los 32, momento a partir del cual decae un poco. El algoritmo *BLISS* por su parte sigue un patrón de dientes de sierra, obteniendo unos mejores resultados para 4, 16 y 64 cores, mientras que para 8 y 32

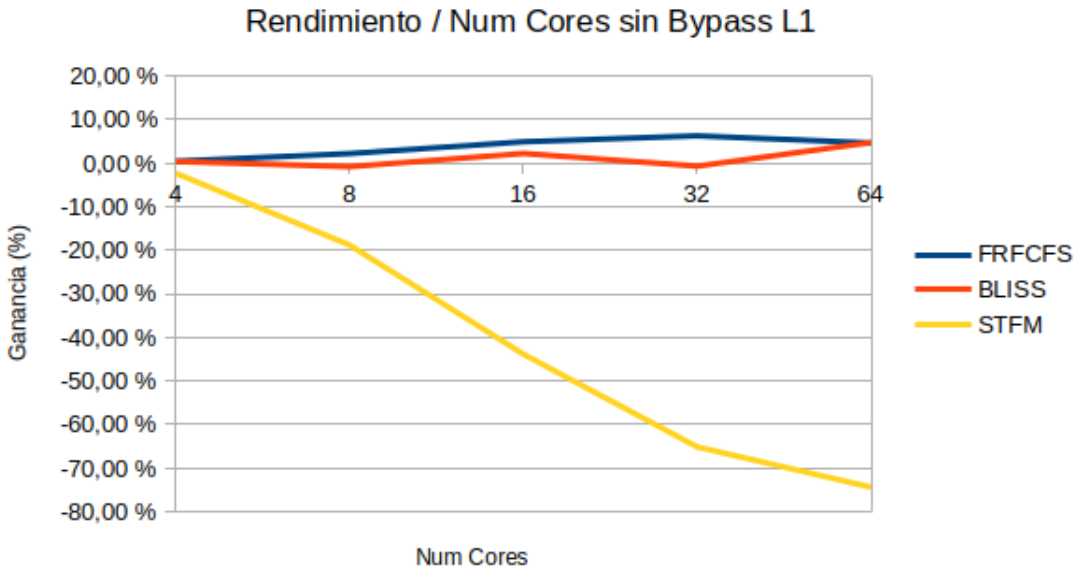

Figura 4.3: Rendimiento por número de cores sin *bypass* de L1

empeora su rendimiento. Por último, para el algoritmo *STFM*, se observa que el rendimiento obtenido es muy malo en comparación con el algoritmo *FIFO*, aunque este valor a priori tan negativo tiene una explicación. La métrica medida en esta gráfica es la ganancia media obtenida para las 13 gráficas simuladas. Para la mayoría de ellas, el rendimiento obtenido es similar al conseguido con los otros algoritmos (en torno a un 5 % de media) pero hay una matriz con la que se comporta especialmente mal, que es *msc01440*.

Para esta matriz, se obtienen valores muy negativos, del orden de -500 % o incluso -1000 % para el caso de 64 cores. Si nos paramos a analizar la estructura de la matriz, podemos hacernos una idea de a qué puede deberse esta diferencia tan grande respecto a otras matrices. Por regla general, las matrices con las que estamos trabajando tienden a estar muy diagonalizadas, es decir, tienen la mayoría de sus datos en la diagonal principal, mientras que el resto de valores son 0. Sin embargo, en el caso de la matriz *msc01440* los datos no están tan concentrados en la diagonal, sino que se distribuyen a lo largo de toda la matriz. Esto puede dar lugar a la diferencia de rendimiento observada tras la realización de las pruebas. La estructura de la matriz mencionada puede observarse en la imagen 3.1 del capítulo 3.

De igual forma, la gráfica 4.4 muestra la ganancia media obtenida en función del número de cores pero esta vez contando con el *bypass* de la cache L1. Se puede apreciar que los resultados siguen la misma tendencia que en la gráfica anterior, a excepción del último algoritmo. En el caso del algoritmo *FRFCFS*, el patrón es prácticamente idéntico al caso sin *bypass* de L1, aumentando según va creciendo el número de cores hasta 32 y, a partir de ese momento, decreciendo un poco. Para *BLISS*, se repite el patrón de dientes de sierra, aunque en este caso las curvas son menos pronunciadas, lo que indica que se comporta de una forma más uniforme. Por último, en el caso del algoritmo *STFM*, la pérdida de rendimiento es igualmente grande, pero a diferencia del caso anterior, la mayor pérdida se produce con 16

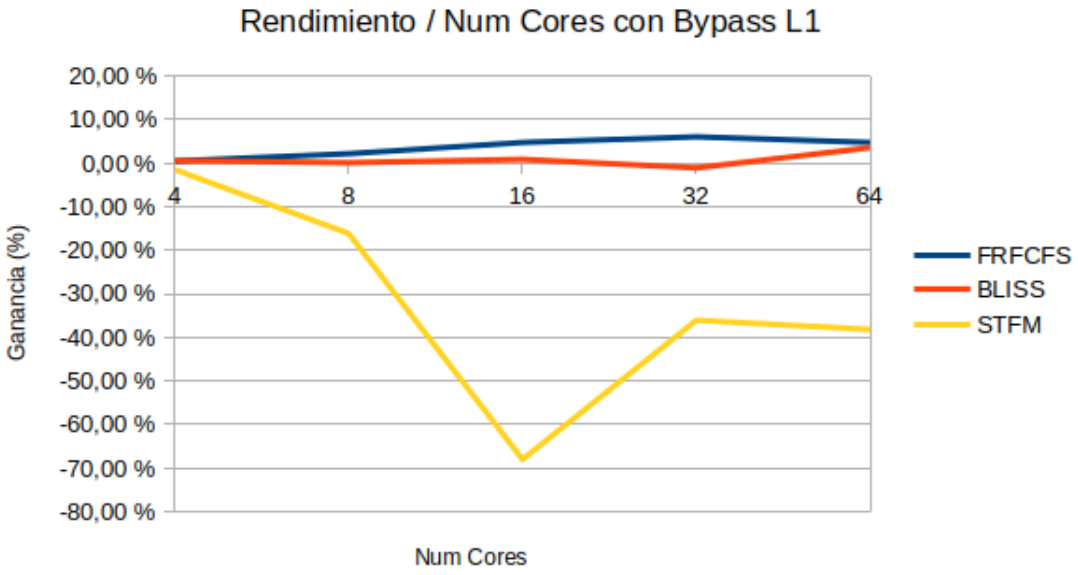

Figura 4.4: Rendimiento por número de cores con *bypass* de L1

cores en vez de 64. El pico de rendimiento negativo es de un -68 % con 16 cores, mientras que para 64 cores aumenta hasta en torno a un -40 %.

Estos valores obtenidos en términos de rendimiento, es posible relacionarlos con la saturación del sistema de memoria. En los experimentos, únicamente se emplea un controlador de memoria, por lo que al ir aumentando el número de cores, se puede observar cómo va aumentando igualmente la presión ejercida sobre el sistema de memoria. Para observar esta saturación, se incluyeron una serie de métricas en el simulador (como se indica en la sección 3.6), de forma que es posible analizar la ocupación de las colas de peticiones.

Para todos los algoritmos, los resultados obtenidos en términos de ocupación son prácticamente idénticos, tanto con *bypass* como sin *bypass*. Para las pruebas realizadas con un número bajo de cores, la mayoría del tiempo la ocupación de la cola es inferior a 5 peticiones, como puede observarse en la gráfica 4.5. Sin embargo, a medida que se va aumentando el número de cores, la composición de cada una de las barras va variando, disminuyendo la zona de color azul correspondiente a 5 peticiones o menos, y aumentando la naranja y verde, correspondientes a los intervalos 5-10 y 10-15 respectivamente.

Finalmente, al alcanzar los 64 cores se puede comprobar la saturación presente en el sistema de memoria, ya que prácticamente la totalidad de las barras que componen la gráfica 4.6 están en color rojo, correspondiente al intervalo +15. Esto significa que el sistema no es capaz de dar servicio a todas las peticiones entrantes en un tiempo adecuado y estas se van acumulando.

Como ya se ha comentado anteriormente, el volumen de datos obtenido es tan grande que no se puede comentar cada caso de forma individual, ya que se superaría con creces la extensión

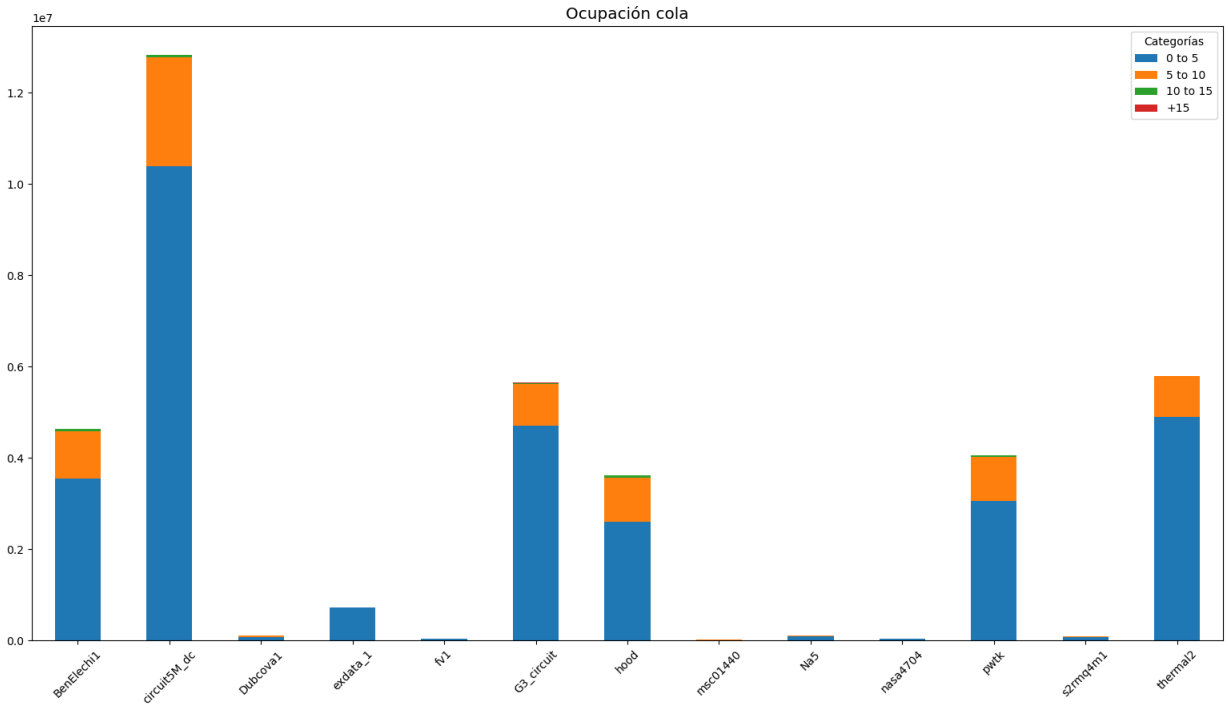

Figura 4.5: Ocupación de la cola de peticiones con FIFO y 4 cores

de este trabajo, pero sí que merece la pena pararse a analizar los casos destacados. Ya se ha comentado el especialmente mal desempeño que obtiene la matriz *msc01440* al utilizar el algoritmo *STFM*, y ahora se va a destacar un caso opuesto.

La matriz *nasa4704* obtiene un muy buen rendimiento con todos los algoritmos, llegando a obtener una ganancia de incluso un 40 % con *STFM* y 64 cores. Esta sobresale respecto a las demás, ya que de media consigue la mayor ganancia de todas las matrices analizadas. Esto lo podemos observar en la gráfica 4.7, dándonos cuenta de que solo obtiene un resultado negativo en una ocasión (*BLISS* con 8 cores) y la aplicación de estos algoritmos con esta carga de trabajo en específico para esta aplicación resulta muy positiva.

Analizados todos los resultados, podemos concluir que, en general, la utilización de estos algoritmos tiene un impacto positivo en el rendimiento del sistema, obteniendo una ganancia de en torno al 5-8 %. Pero, como se ha comprobado, su utilización no siempre supone una mejora, ya que hay ocasiones en las que no se consigue ninguna ventaja o, incluso, se producen pérdidas de rendimiento. Es necesario analizar cada caso de forma independiente y optar por emplear la técnica que mejor se ajuste a sus necesidades.

A priori, los resultados pueden no parecer muy sorprendentes ya que las ganancias obtenidas no llegan a superar el 10 % de media pero, si se analiza con detenimiento, esta ganancia se está obteniendo únicamente a través del reordenado de peticiones, sin incluir ninguna ayuda *hardware* ni ningún otro tipo de mejora, por lo que se trata de una cifra relativamente alta.

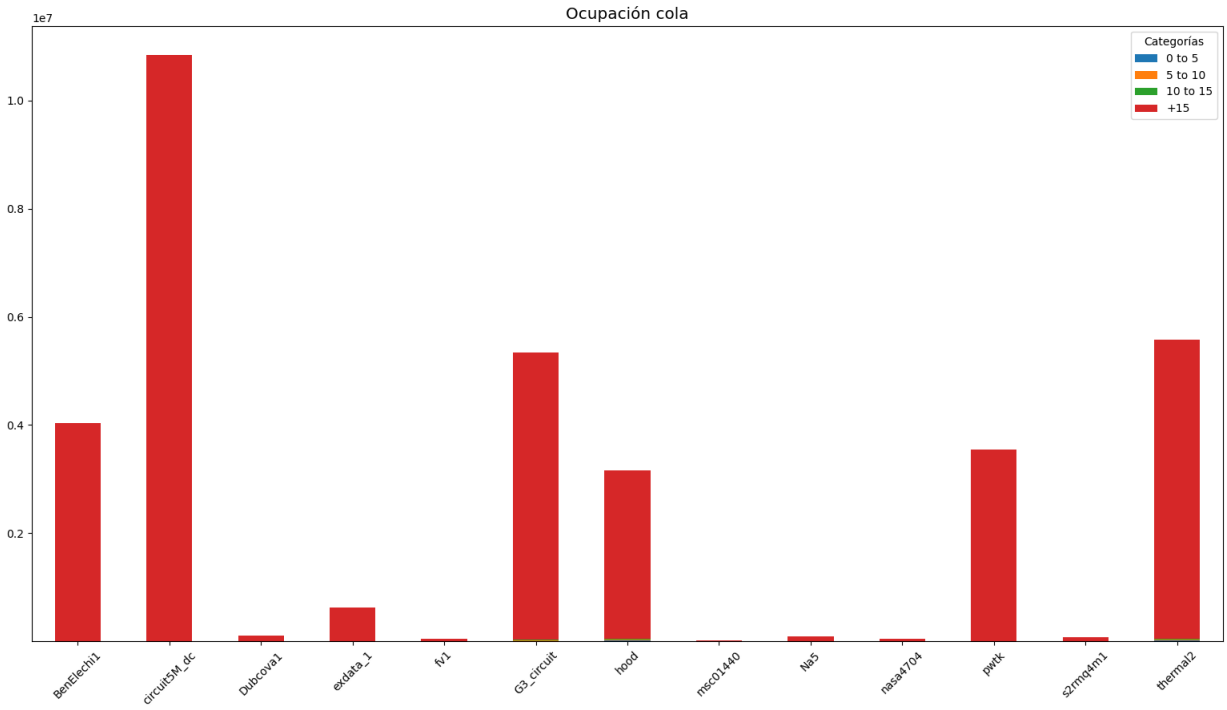

Figura 4.6: Ocupación de la cola de peticiones con FIFO y 64 cores

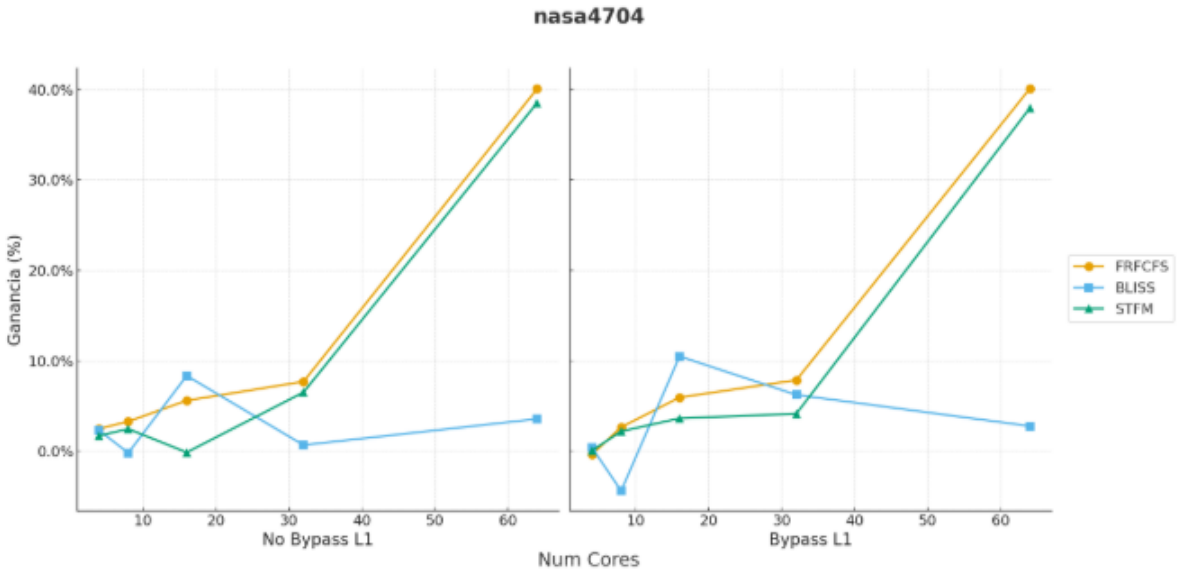

Figura 4.7: Rendimiento por número de cores para la matriz *nasa4704*

Capítulo 5

Conclusiones

En este capítulo final se describen los objetivos conseguidos tras la realización de este proyecto y, a su vez, las posibles líneas de mejora que se podrían implementar en el simulador *Coyote*, tanto en lo referido a planificación de peticiones como en otros aspectos de la herramienta, en el futuro.

5.1. Objetivos conseguidos

Una vez finalizado este trabajo, resulta relevante hacer una vista en retrospectiva de los objetivos iniciales descritos en la Sección 1.2. A continuación, se hace un mapeo entre los logros conseguidos y los objetivos propuestos:

- *Adición de tres nuevos algoritmos de planificación en la herramienta*: El simulador ha sido modificado con éxito para ser capaz de incluir los tres algoritmos de planificación de peticiones a memoria propuestos en el trabajo. Estos algoritmos han sido implementados continuando con la modularidad de la herramienta, de forma que no se han modificado en gran medida las capacidades previamente existentes y se simplifica la posibilidad de incorporar nuevos algoritmos en el futuro. Este avance dota a la herramienta de una mayor flexibilidad a la hora de realizar las simulaciones ya que se dispone de más opciones para elegir como configurar el controlador de memoria.

  También se han conseguido modificar algunas áreas de la herramienta para conseguir dotar de una mayor visión del sistema al controlador de memoria. Con los cambios realizados, el controlador es capaz de conocer algunos parámetros que pueden ser importantes a la hora de realizar la simulación y que antes, por el tipo de algoritmos disponibles, no era necesario conocer.

- *Comparativa de algoritmos*: En base a los resultados obtenidos en el capítulo 4, podemos extraer diferentes conclusiones. En primer lugar, como se comentó en la sección

2.5, el problema de la planificación tiene mucha complejidad y, por tanto, no existe una solución que sea óptima en cualquier situación, si no que, en función del objetivo definido, puede haber alguna que obtenga un mayor rendimiento para una determinada métrica. Se ha comprobado, gracias a la batería de diferentes matrices, que los algoritmos tienen un rendimiento diferente en función de cual sea la estructura empleada. En determinadas situaciones alguno de los algoritmos se comporta especialmente bien y consigue una mejora en el rendimiento notable, mientras que en otras situaciones la mejora puede ser mínima o, incluso, producirse pérdidas de rendimiento. Pero teniendo en cuenta los valores promedios los resultados son positivos.

Estos elementos permiten satisfacer el propósito fundamental del desarrollo de este proyecto, el cual era el estudio de propuestas de planificación de peticiones de acceso a memoria en entornos multicore basados en arquitectura *RISC-V* vectorial. Se han estudiado e implementado tres nuevos algoritmos, y se ha contrastado su rendimiento en el simulador *Coyote* frente a otros algoritmos disponibles en la plataforma.

5.2. Trabajos Futuros

A lo largo del desarrollo de este proyecto han surgido nuevas oportunidades de mejora para la herramienta. Sin embargo, debido a limitaciones de tiempo, no ha sido posible implementarlas. Estas mejoras quedan abiertas para una posible implementación futura y se detallan a continuación:

- *Implementación de nuevos algoritmos*: Tras la realización de este proyecto, han quedado correctamente implementados en el controlador de memoria del simulador los algoritmos *FRFCFS*, *BLISS* y *STFM*; y se han podido comparar para analizar las ventajas y desventajas de cada uno de ellos. Pero para dotar de una mayor flexibilidad a las simulaciones, se plantea la posibilidad de continuar añadiendo nuevos algoritmos que intenten conseguir el máximo rendimiento del sistema, o se centren en maximizar el resultado para una métrica objetivo, como pueden ser algunos de los mencionados en la sección 2.5 u otros diferentes.
- *Mejoras en otras áreas de la herramienta*: El ámbito de acción de este proyecto se ha reducido principalmente al sistema de memoria, más concretamente el controlador de memoria. Sin embargo, en el simulador se modelan muchos otros componentes que permiten replicar un sistema completo, y cada uno de ellos es susceptible de poderse mejorar. Un área que podría ser interesante estudiar sería el sistema de interconexión, permitiendo modelar diferentes tecnologías con diferentes velocidades y anchos de banda, lo que dotaría al simulador de un mayor realismo.
- *Expansión a nuevas tecnologías*: De cara al futuro, el mundo de la memoria evolucionará con nuevas tecnologías y la herramienta debe estar preparada para integrarlas y

poder realizar las simulaciones de forma efectiva. Esto resulta esencial para mantener *Coyote* actualizado y que se pueda seguir utilizando de acuerdo a las necesidades actuales y futuras. En el final de la sección 2.2 se indican algunas tecnologías que ya se están comenzando a implementar a día de hoy en productos comerciales y que podría resultar interesante modelar. Es muy importante mantener el sistema modular para poder acoplar las nuevas funcionalidades de la forma más sencilla posible, sin necesidad de modificar en gran medida las capacidades ya existentes. Esta mejora dotaría de una mayor flexibilidad y veracidad a la herramienta ya que la permitiría alinearse con las tendencias tanto académicas como comerciales.

Bibliografía

[1] Philip Machanick. Approaches to addressing the memory wall. 11 2002.

[2] Rajeev Balasubramonian. *Innovations in the Memory System*. Morgan Claypool Publishers, 2019.

[3] Wolfgang Ertel. *Introduction to Artificial Intelligence, Third Edition*. Undergraduate Topics in Computer Science. Springer, 2025.

[4] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, 2017.

[5] J. Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011.

[6] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. *A Modern Primer on Processing in Memory*, pages 171–243. 01 2023.

[7] Boria Perez, Alexander Fell, and John D. Davis. Coyote: An open source simulation tool to enable risc- v in hpc. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 130–135, 2021.

[8] Chi Zhang, Paul Scheffler, Thomas Benz, Matteo Perotti, and Luca Benini. Near-memory parallel indexing and coalescing: Enabling highly efficient indirect access for spmv. In *2024 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2024.

[9] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. 28(2):128–138, 2000.

[10] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. Bliss: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):3071–3087, 2016.

[11] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160, 2007.

[12] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[13] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

[14] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[15] Josué Feliu Pérez. Planificación de tareas considerando la contención en la jerarquía de memoria de procesadores multinúcleo. 2013.

[16] Paula Navarro Alfonso. Diseño de controladores de memoria eficientes para futuros sistemas. 2015.

[17] Satoshi Yamada and Shigeru Kusakabe. Effect of context aware scheduler on tlb. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[18] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. *SIGARCH Comput. Archit. News*, 36(3):39–50, Jun 2008.

[19] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[20] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *2008 International Symposium on Computer Architecture*, pages 63–74, 2008.

[21] Ruth Malan, Dana Bredemeyer, et al. Functional requirements and use cases. *Bredemeyer Consulting*, 2001.

[22] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.

[23] Texas A&M University. Suitesparse matrix collection. `https://sparse.tamu.edu`. Último acceso: Mayo de 2025.

[24] BSC. Coyote. `https://github.com/MEEPproject/coyote.git`. Último acceso: Mayo de 2024.

Apéndice A

Códigos realizados

```python
import scipy.io as sio
import scipy.sparse as sp

if __name__ == '__main__':
    data = sio.loadmat('thermal2.mat')['Problem']['A']
    matriz_csc = sp.csc_matrix(data[0,0])

    valores = []
    for i in range(matriz_csc.indptr[-1]):
        valores.append(1)

    with open('dataset.h', 'w') as archivo:
        archivo.write('#ifndef __DATASET_H\n')
        archivo.write('#define __DATASET_H\n\n')
        archivo.write(f'#define NROWS {matriz_csc.shape[0]}\n\n')
        archivo.write(f'#define N_NON_ZERO {matriz_csc.indptr[-1]}\n\n')
        archivo.write('static long ia[N_ROWS+1] __attribute__((aligned (1024)))=\n')
        archivo.write('{\n')
        archivo.write(', '.join(map(str, matriz_csc.indptr)))
        archivo.write('\n};\n\n')
        archivo.write('static long ja[N_NON_ZERO] __attribute__((aligned (1024)))=\n')
        archivo.write('{\n')
        archivo.write(', '.join(map(str, matriz_csc.indices)))
        archivo.write('\n};\n\n')
        archivo.write('static double a[N_NON_ZERO] __attribute__((aligned (1024)))=\n')
        archivo.write('{\n')
        archivo.write(', '.join(map(str, matriz_csc.data.astype(int))))
        archivo.write('\n};\n\n')
        archivo.write('static double x[N_NON_ZERO] __attribute__((aligned (1024)))=\n')
        archivo.write('{\n')
        archivo.write(', '.join(map(str, valores)))
        archivo.write('\n};\n\n\n')
        archivo.write('#endif //__DATASET_H')
```

Función 1: Script Python para transformar las matrices en el formato adecuado *spmv-vec*

```
1         int chunk=nrows/ncores;
2         int start=coreid*(chunk);
3         int end = (coreid==ncores-1) ?  nrows : start+chunk;
4         for (int row = start; row < end; row++) {
5             int nnz_row = ia[row + 1] - ia[row];
6             int rvl, gvl;         // requested & granted vector lengths
7             int idx = ia[row];
8             y[row]=0.0;
9             for(int colid=0; colid<nnz_row; colid +=gvl )
10            {    //blocking on MAXVL
11                rvl = nnz_row - colid;
12                gvl = __builtin_epi_vsetvl(rvl, __epi_e64, __epi_m1);
13                __epi_1xf64 va = __builtin_epi_vload_1xf64(&a[idx+colid], gvl);
14                __epi_1xi64 v_idx_row = __builtin_epi_vload_1xi64(&ja[idx+colid], gvl);
15                __epi_1xi64 vthree = _BSCAST_i64(3, gvl);
16                v_idx_row = __builtin_epi_vsll_1xi64(v_idx_row, vthree, gvl);
17                __epi_1xf64 vx = __builtin_epi_vload_indexed_1xf64(x, v_idx_row, gvl);
18                __epi_1xf64 vprod = __builtin_epi_vfmul_1xf64(va, vx, gvl);
19                __epi_1xf64 partial_res = _BSCAST_f64(0.0, gvl);
20                partial_res = __builtin_epi_vfredsum_1xf64(vprod, partial_res, gvl);
21                y[row] += __builtin_epi_vfmv_f_s_1xf64(partial_res);
22            }
23          }
```

Función 2: SpMV básico

```
int k=1;
int chunk=nrows/(ncores*k);
for(int i = 0; i < k; i++){
  int start = coreid*chunk+i*chunk*ncores;
  int end = (coreid==ncores-1 && i==k-1) ? nrows : start+chunk;
  for (int row = start; row < end; row++) {
    int nnz_row = ia[row + 1] - ia[row];
    int rvl, gvl;        // requested & granted vector lengths
    int idx = ia[row];
    y[row]=0.0;
    for(int colid=0; colid<nnz_row; colid +=gvl )
    {    //blocking on MAXVL
        rvl = nnz_row - colid;
        gvl = __builtin_epi_vsetvl(rvl, __epi_e64, __epi_m1);
        __epi_1xf64 va = __builtin_epi_vload_1xf64(&a[idx+colid], gvl);
        __epi_1xi64 v_idx_row = __builtin_epi_vload_1xi64(&ja[idx+colid], gvl);
        __epi_1xi64 vthree = _BSCAST_i64(3, gvl);
        v_idx_row = __builtin_epi_vsll_1xi64(v_idx_row, vthree, gvl);
        __epi_1xf64 vx = __builtin_epi_vload_indexed_1xf64(x, v_idx_row, gvl);
        __epi_1xf64 vprod = __builtin_epi_vfmul_1xf64(va, vx, gvl);
        __epi_1xf64 partial_res = _BSCAST_f64(0.0, gvl);
        partial_res = __builtin_epi_vfredsum_1xf64(vprod, partial_res, gvl);
        y[row] += __builtin_epi_vfmv_f_s_1xf64(partial_res);
    }
  }
}
```

Función 3: SpMV con granularidad

Apéndice B

Configuración simulador

Parámetro	Valor
meta.params.architecture	tiled
meta.params.simulation_mode	execution-driven
meta.params.vector_bypass_l1	true/false
meta.params.vector_bypass_l2	false
meta.params.l1_writeback	false
meta.params.fast_cache	false
meta.params.cmd	../../apps/spmv-vec/spmv
top.arch.params.frequency_ghz	1
top.arch.params.num_cores	4/8/16/32/64
top.arch.params.num_threads_per_core	1
top.arch.num_tiles	1
top.arch.num_memory_cpus	1
top.arch.isa	RV64IMAFDCV
top.arch.icache_config	32:8:64
top.arch.dcache_config	32:8:64
top.arch.varch	v1024:e64:s1024
top.arch.lanes_per_vpu	16
top.arch.tile.params.num_l2_banks	32
top.arch.tile.params.latency	1
top.arch.tile.params.l2_sharing_mode	fully_shared
top.arch.tile.params.bank_policy	set_interleaving
top.arch.tile.params.tile_policy	set_interleaving
top.arch.tile.arbiter.params.q_sz	16
top.arch.tile.l2_bank.params.line_size	64
top.arch.tile.l2_bank.params.size_kb	32
top.arch.tile.l2_bank.params.associativity	16
top.arch.tile.l2_bank.params.always_hit	false
top.arch.tile.l2_bank.params.writeback	false
top.arch.tile.l2_bank.params.miss_latency	15
top.arch.tile.l2_bank.params.hit_latency	15
top.arch.memory_cpu.params.line_size	64
top.arch.memory_cpu.params.latency	1
top.arch.memory_cpu.params.num_llc_banks	16
top.arch.memory_cpu.llc.params.line_size	64
top.arch.memory_cpu.llc.params.size_kb	512
top.arch.memory_cpu.llc.params.associativity	16
top.arch.memory_cpu.llc.params.always_hit	false
top.arch.memory_cpu.llc.params.miss_latency	15
top.arch.memory_cpu.llc.params.hit_latency	15
top.arch.memory_controller.params.num_banks	32
top.arch.memory_controller.params.num_banks_per_group	4
top.arch.memory_controller.params.request_reordering_policy	fifo/frfcfs/bliss/stfm
top.arch.memory_controller.params.command_reordering_policy	fifo

Cuadro B.1: Parámetros para el fichero simple_arch.yml