UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Análisis de requerimientos y diseño de un controlador de memoria principal no volátil

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Miguel Antonio Avargues Gutiérrez

Tutor: Julio Sahuquillo Borrás

Segundo tutor: Salvador Vicente Petit Martí

Curso 2020-2021

Dedicatoria

Me gustaría dedicarle este TFG a mis padres, María y Jacinto, por ser el faro en el que confiar para guiarme con certeza.

A mi hermana Maite, por ser esa persona que se preocupa por todos, antes incluso que ella misma.

A mis dos tías, Pepa y Antonia, por el increíble apoyo que ofrecen tanto a mi como al resto de mi familia.

A mi abuela Teresa, que lleva ahí desde que tengo memoria y siempre consigue sacarme una sonrisa.

Agradecimientos

Agradecer a Julio y Salvador el excepcional trabajo de tutoría que han realizado. No solo revisando la memoria, sino a lo largo de todo el proceso ayudando a encontrar soluciones a problemas que me parecían infranqueables.

A mi tio Antonio y el resto de familiares que han ayudado a revisar este TFG.

A Manel y el resto de compañeros del grupo por ayudarme con todas las dudas y problemas que he tenido en la realización de este TFG.

A todos mis amigos, por haber ayudado a ese Miguel frustrado que ha aparecido ya no solo a lo largo de la elaboración de todo el TFG, sino de toda su vida académica.

Glosario

Benchmark (también *benchmarks*) Programa que se utiliza con la finalidad de obtener métricas de prestaciones del sistema en el que se ejecuta. 24, 27, 28, 41–44

Benchmarks Véase *benchmark*. 3, 13, 17, 23, 24, 26, 27, 29, 41, 43, 48

Clústeres Conjunto de sistemas informáticos que pueden trabajar de forma colaborativa con tal de ejecutar grandes cargas de trabajo. Como una tarea muy pesada o muchas más ligeras. 7

Cores Los *cores* o núcleos, de la unidad central de procesamiento habilitan a esta a ejecutar diversas instrucciones al mismo tiempo. 5, 9

CPU La *central processing unit*, también conocida CPU por sus siglas y traducida como unidad central de procesamiento, es aquella parte encargada de ejecutar las instrucciones de un sistema informático. 5, 6, 13, 15, 16, 23, 42

DIMM Los *Dual In-line Memory Module* son el estándar de organización para los *ranks* de memoria principal. En función de la cantidad de conexiones nos encontramos con diferentes revisiones del mismo estándar. 1, 2, 9–11, 13, 47

DRAM *Dynamic Random Access Memory* o memoria dinámica de acceso aleatorio. Este tipo de memorias suele ser el tipo de memoria principal más común actualmente en los sistemas informáticos, ofreciendo una baja latencia de acceso a cambio de una baja densidad de almacenamiento. 1–3, 5, 7, 9–11, 13, 14, 17, 45, 49

HPC *High Performance Computing*, traducido como computación de altas prestaciones y se conocen así a los sistemas informáticos dedicados al tratamiento masivo de datos o ejecución de cargas de trabajo muy pesadas, por ejemplo, simulaciones en el ámbito científico. 2, 47

ISA *Instruction Set Architecture* o arquitectura del conjunto de instrucciones es el conjunto de instrucciones que es capaz de ejecutar un procesador. 14, 29

NV *Non-volatile* o no volátil, en lo referente a este trabajo, es aquella tecnología encargada de implementar memorias cuyos datos no se eliminan al dejar de ofrecerle corriente eléctrica. 5, 7

NVMM *Non-volatile Main Memory* o memoria principal no volátil es el uso de la tecnología no volátil como tecnología de almacenamiento para memoria principal. 1, 2, 5, 12, 33, 47

NVRAM *Non-volatile Random Access Memory* o memoria no volátil de acceso aleatorio. Este tipo de memoria es más novedosa a la hora de usarse como memoria principal. Usada fundamentalmente en servidores ofrecen una densidad de almacenamiento mucho mayor a DRAM a coste de unas latencias más altas. Existen varios tipo de implementaciones de este tipo de memorias. 1–3, 7, 12–14, 43, 45, 47–49

path El *path* es la ubicación de un fichero o directorio en un sistema de archivos. 16, 20, 23

pipeline La *pipeline* de un procesador es el conjunto de fases en las que está divido. Cada una de estas fases ejecuta una parte diferente de las instrucciones, con el objetivo de poder ejecutar varias instrucciones al mismo tiempo en diferentes fases.. 14, 15

SaaS El *Software as a service*, traducido como *software* como servicio es todo aquel que ofrece algún tipo de aplicación a cambio de algún tipo de retribución, normalmente, una subscripción. 1

Script (también ***scripts*****)** Conjunto de instrucciones, normalmente pequeño, que se encarga de realizar una tarea con tal de evitar repeticiones. 26–28, 35

Scripts Véase *script*. 3, 15, 16, 41, 43

Speedup El *speedup* es un valor que mide la ganancia de prestaciones entre dos sistemas. 41, 42

Resum

En l'actualitat a causa de la descentralització de la computació, la majoria dels càlculs es realitzen en servidors que executen càrregues de treball pesades. Usualment, aquests servidors executen aplicacions que fan un ús elevat de memòria principal. Això implica que les memòries DRAM convencionals no emmagatzemen totes les dades necessàries amb tal d'executar aquestes aplicacions, sent açò el principal coll de botella en aquests sistemes. Les memòries NVRAM intenten solucionar aquest problema oferint una major densitat de emmagatzenament d'informació a canvi de major latències d'accés. En aquest projecte s'implementa un controlador de memòria principal per a memòries NVRAM amb l'objectiu de millorar les prestacions d'aplicacions que accedeixen a aquests tipus de memories.

Paraules clau: Memòria no volàtil; Memòria principal; Controlador de memòria; NVRAM; NVMain; Gem5

Resumen

En la actualidad debido a la descentralización de la computación, la mayoría de los cálculos se realizan en servidores que ejecutan cargas de trabajo pesadas. Usualmente, estos servidores ejecutan aplicaciones que hacen elevado uso de memoria principal. Esto conlleva que las memorias DRAM convencionales no almacenan todos los datos necesarios para la ejecución de estas aplicaciones, siendo así el principal cuello de botella en estos sistemas. Las memorias NVRAM intentan solucionar este problema ofreciendo mayor densidad de almacenamiento de información a coste de latencias de acceso mayores. En este proyecto se implementa un controlador de memoria principal para memorias NVRAM con el objetivo de mejorar las prestaciones de aplicaciones que acceden a este tipo de memorias.

Palabras clave: Memoria no volátil; Memoria principal; Controlador de memoria; NVRAM; NVMain; Gem5

Abstract

Currently due to the computation decentralization, most of the computing is made on servers who perform heavy workloads. Usually, these servers run applications which make high use of main memory. This fact makes conventional DRAM memories unable to store all data needed in order to run these applications, making them the main bottleneck on those systems. NVRAM memories try to solve this bottleneck by offering more memory storage density at the cost of higher access latency. In this project a memory controller for NVRAM memories is implemented, with the goal of increasing the performance of applications that access these types of memories.

Key words: Non-volatile memory; Main memory; Memory controller; NVRAM; NVMain; Gem5

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

1.1 Descripción del problema

En la actualidad la descentralización de la computación permite realizar prácticamente cualquier cómputo desde dispositivos con muy poca potencia de cálculo pero que disponen de una conexión a Internet. Gracias a estas características, pueden entrar en contacto con servidores que les proporcionan una gran cantidad de servicios, teniendo que hacer poco más que mostrar una interfaz y tratar los paquetes de respuesta de los servidores. Este movimiento de la carga de computación desde los dispositivos usuarios hacia los servidores lleva en auge desde hace una década. Por ejemplo, con servicios como «Dropbox» o «Google Drive». Estos no eran mas que el principio de una oleada de servicios en la nube conocidos como *software as a service* o SaaS.

Esta explosión de servicios en la nube y la correspondiente movilización de la carga de trabajo a los servidores, lleva a que los sistemas que se encargan de la computación real, los servidores, necesiten conseguir altas prestaciones para poder dar respuesta a las peticiones que se realizan. Normalmente, las aplicaciones que se ejecutan en estos servidores necesitan una gran cantidad de datos que se deben almacenar en la memoria principal. Estas cantidades de datos continúan aumentando, por lo que la capacidad de almacenamiento de memoria principal se convierte en un factor clave para las prestaciones. Este hecho se convierte en un problema tecnológico puesto que la memoria principal, actualmente implementada con tecnología *Dynamic Random-Access Memory* o DRAM, presenta problemas de escalabilidad. En efecto, la tecnología DRAM ha escalado (en Mbits/chip) en un factor de dos veces cada año y medio desde 1985. Sin embargo, esta tendencia se ha ralentizado desde principios de este siglo y es un desafío para una tecnología de nodo inferior a 10nm ya que la implementación presenta problemas de fiabilidad [1]. En consecuencia, los actuales módulos de memoria principal DIMM tienen una capacidad de almacenamiento relativamente limitada para los requerimientos de datos de las aplicaciones actuales.

Para atacar este problema, han aparecido recientemente las memorias *Non Volatile Random Access Memory* o NVRAM, también conocidas como NVMM, siglas de *Non Volatile Main Memory*. Estas memorias presentan una capacidad de almacenamiento muy

superior a las memorias DRAM convencionales, menor coste por bit y con persistencia de datos; es decir, la información almacenada no se pierde cuando se interrumpe la corriente. Las NVRAM se plantean como una gran opción de futuro para convertirse en una tecnología candidata para implementar memoria principal o bien situarse por debajo de la memoria DRAM en la jerarquía de memoria.

Si se utiliza la memoria NVRAM como memoria principal del sistema, comparada con la memoria DRAM, presenta ciertas ventajas e inconvenientes [2]. La principal ventaja es la mayor capacidad de estas, que permite reducir el número de fallos de página que implican un acceso a la memoria secundaria mucho más lenta. El inconveniente es que el tiempo de acceso es mayor que en el caso de la DRAM. Por tanto existe un compromiso que debe resolverse para alcanzar buenas prestaciones. Este problema de la mayor latencia es el que queremos analizar e intentar resolver mediante el diseño de un novedoso controlador de memoria que sea capaz de reducir los tiempos de acceso a unos más parecidos a aquellos de una memoria principal con DRAM DIMM.

1.2 Objetivos

El presente TFG se enmarca dentro de un proyecto mucho más ambicioso que persigue diseñar un controlador de memoria principal híbrida, compuesto tanto por memoria DRAM como NVRAM, que abarque lo mejor de ambas tecnologías: la rapidez de la DRAM y la capacidad de almacenamiento de la NVRAM. Para ello, y como primer paso, el presente TFG se centra en memoria NVRAM y los simuladores que se usarán a lo largo de este proyecto. En concreto, el TFG persigue los siguientes objetivos:

- Familiarizarse con la simulación de sistemas con memoria NVMM, entendiendo el código que modela su funcionamiento y modificándolo de acuerdo con los restantes objetivos del proyecto.
- Elaborar y diseñar un controlador de memoria principal para NVRAM DIMM que incluya una memoria cache para mejorar las prestaciones.
- Analizar el comportamiento y requerimientos de las aplicaciones que utilizan un gran volumen de datos usadas en servidores de gran potencia de cómputo, también conocidos como servidores HPC. Este estudio cubre la obtención de trazas de memoria y la extracción de estadísticas para distintas aplicaciones.
- Analizar las prestaciones del controlador diseñado frente a un controlador de memoria NVRAM convencional.

1.3 Estructura de la memoria

Para alcanzar los objetivos mencionados, este TFG se ha estructurado en siete capítulos:

- En el capítulo 2 se comenta de forma breve, a modo de contexto tecnológico, la jerarquía de memoria y se entra en algo más de detalle sobre las memorias caché; a continuación se expone el funcionamiento genérico de un controlador de memoria; seguidamente se expone el estado actual de las tecnologías DRAM y NVRAM. Tras la explicación de estos conceptos, se presentan simuladores de memoria principal con memorias no volátiles, y simuladores de procesadores ciclo-a-ciclo ampliamente utilizados por la comunidad científica. También se nombran los simuladores que vamos a usar en el proyecto.

- El capítulo 3 presenta el entorno experimental con sus características, simuladores y las cargas de trabajo (*benchmarks*). Además, se describe el proceso para llevar a cabo la simulación y preparar el entorno para la realización de esta.

- El capítulo 4 describe la forma en la que se modelan los controladores de memoria en el entorno usado. Asimismo, se introduce la propuesta que vamos a implementar y se explican los pasos necesarios para la realización de esta.

- En el capítulo 5 se presentan los resultados de los *benchmarks* ejecutados en el sistema simulado. También se realiza un análisis de trazas mediante *scripts* personalizados.

- Finalmente, en el capítulo 6 se resumen las principales conclusiones que hemos extraído durante la realización de este TFG. También se comenta la relación con los estudios cursados y futuro trabajo a realizar.

CAPÍTULO 2

Estado del arte: contexto tecnológico y simuladores

En el presente capítulo se presentan algunos artículos de interés respecto al tema de este proyecto, así como algunos elementos necesarios para facilitar la comprensión de este TFG. Para ello, dividimos el capítulo en dos partes. La primera introduce la jerarquía de memoria, las memorias caché, los controladores de memoria y las tecnología DRAM y NV. En la segunda parte se discuten simuladores de memoria NVMM recientes, así como simuladores de procesadores actuales que modelan con detalle la parte correspondiente al núcleo computacional. Tras el estudio de estos simuladores, se procede a la elección del mejor simulador de cada tipo para llevar a cabo el proyecto.

2.1 Contexto tecnológico

2.1.1. Conceptos previos: Jerarquía de memoria

En el estado actual de la computación, la velocidad de la unidad central de procesamiento, también conocida como CPU, no es el único factor vital en las prestaciones de los sistemas. De hecho, la velocidad con la que la jerarquía de memoria ofrece datos e instrucciones a la CPU y sus núcleos —o *cores*—, es un factor crítico en las prestaciones globales del sistema [3], y a menudo es el principal cuello de botella. Si bien, este problema ha aparecido numerosas veces a lo largo de la evolución de los procesadores y fue acuñado en 1995 como *memory wall* [3], en la actualidad el problema es mucho mayor debido al continuo aumento en la disparidad de prestaciones entre ambos elementos; por ejemplo, el procesador da soporte a la ejecución de muchas más aplicaciones, lo que aumenta la contención en memoria principal y por tanto el tiempo de espera para acceder a esta.

Es por ello que para intentar mitigar este «muro» en la velocidad de las memorias se añaden memorias con distintas tecnologías a lo largo de la jerarquía, para complementar las carencias de unas con las otras. Desde muy rápidas pero con poca capacidad de almacenamiento (p.e. las cachés del primer nivel de la jerarquía) hasta dispositivos con gran capacidad pero varios órdenes de magnitud más lentos que las cachés del procesador.

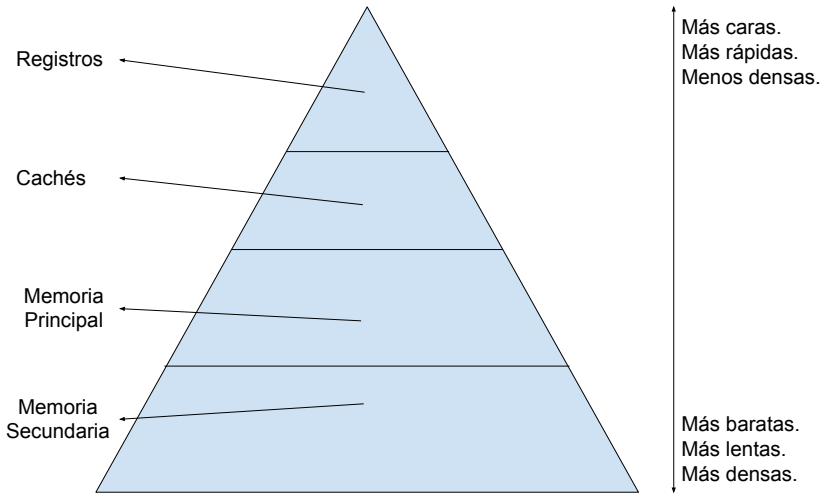

Figura 2.1: Partes de la jerarquía de memoria.

A continuación, y para hacer este proyecto autocontenido, se presenta una breve introducción la jerarquía de memoria.

La figura 2.1 muestra una jerarquía de memoria típica estructurada en cuatro niveles, empezando por la memoria secundaría, pasando por la memoria principal, los distintos niveles de cachés y hasta los registros. A continuación, se describen estos niveles:

- Registros: memorias con una capacidad de almacenamiento de pocos bytes que tiene el procesador donde se almacena un pequeño conjunto de datos (normalmente del ancho de palabra del procesador), sobre las que la CPU realiza computaciones y donde posteriormente guarda el resultado. El objetivo de estas memorias es ofrecer una alta velocidad a la unidad aritmético-lógica para no afectar negativamente a la frecuencia del procesador o incrementar excesivamente el número de etapas de segmentación.

- Cachés: memorias cuya capacidad de almacenamiento es del orden de KB o MB y que se suelen implementar en el mismo chip que el procesador. Normalmente hay tres niveles, pero en ocasiones la jerarquía de cache puede llegar a tener hasta cuatro niveles. Cada uno de estos niveles es más rápido que el siguiente pero dispone de menos espacio de almacenamiento. La finalidad de estas memorias es intentar esconder el tiempo de acceso de otras memorias off-chip que disponen de más datos. Esto se consigue gracias a los principios de localidad temporal (si se accede a un dato, es probable que se vuelva a usar dentro de poco) y localidad espacial (si un dato se acaba de usar, es probable que los que están en una dirección cercana a este se vayan a usar).

- Memoria principal: memoria del orden de GB donde se debe tener toda la información de los programas que se están ejecutando. Además, almacena información de

otros dispositivos del sistema como la tarjeta gráfica. En sistemas de altas prestaciones como clústeres -que son un conjunto de sistemas informáticos que pueden trabajar de forma colaborativa con tal de ejecutar grandes cargas de trabajo. Como una tarea muy pesada o muchas más ligeras- no es extraño encontrarse TB de almacenamiento para poder ofrecer suficiente memoria a cada uno de los núcleos de los procesadores. Este es el nivel en el que se encuentran las memorias NVRAM. Actualmente, la tecnología en la que se implementa es DRAM aunque ya existen variaciones como Intel© Optane™ que son implementaciones con la tecnología NV.

- Memoria secundaria: este nivel de la jerarquía es el que presenta una mayor densidad de almacenamiento. Capaz de almacenar varios TB en un único dispositivo. En este nivel se dispone de dispositivos con la tecnología NV. Estos son conocidos como NVM express o simplemente «NVMe» y los discos duros de estado sólido o SSD por sus signas en inglés. En contraposición a los discos duros más tradicionales, ofrecen un ancho de banda mucho mayor con una densidad de integración similar. La diferencia principal entre los discos «NVMe» y los SSD es a qué buses se conectan. Los discos «NVMe» se conectan al bus PCI, mientras que los de estado sólido se conectan al bus SATA, más lento que el PCI.

2.1.2. Memorias caché

Como se ha comentado previamente, las cachés son memorias cuya capacidad varía de decenas de KB hasta decenas de MB. La unidad de almacenamiento en la cache es un bloque o conjunto de datos, generalmente, de 64 B. Cuando el procesador referencia un dato, el bloque completo de 64 B que lo contiene se trae de memoria principal y se almacena en la cache. Las memorias caché tienen latencias de acceso relativamente bajas y ayudan a mejorar las prestaciones gracias a explotar los principios de localidad temporal y espacial que exhiben las aplicaciones. En otras palabras, las prestaciones mejoran porque accesos posteriores a los datos almacenados tienen una latencia mucho menor que acceder a memoria principal.

En general, las cachés funcionan de la siguiente forma. A partir de la dirección física del dato que se desea leer o escribir se extraen los campos «etiqueta» e «índice». El campo «índice» se utiliza como entrada a un decodificador para seleccionar el conjunto a acceder. Esto es así porque una caché hardware convencional se organiza en conjuntos donde pueden almacenarse uno o varios bloques. El número máximo de bloques que pueden almacenarse por conjunto se conoce como el número de vías o grado de asociatividad.

El campo «etiqueta» identifica de forma única a un bloque de memoria. Esta etiqueta se utiliza para comprobar que la dirección del bloque almacenado coincide con el solicitado. Si la caché contiene los datos se dice que se ha producido un acierto de caché. Sin embargo, cuando no los tiene, la situación se conoce como un fallo de caché. En este caso sería necesario traer de memoria principal el bloque que contiene el dato referenciado a caché.

Por otro lado, cuando se almacena un bloque de datos en una vía de un conjunto, también se almacenan unos bits adicionales. Estos bits contienen información para los algoritmos de reemplazo o para otros usos.

Atendiendo al número de vías por conjunto se pueden diferenciar entre tres tipos de políticas de ubicación que afectan a la organización de la memoria caché y determinan el lugar donde pueden colocarse los bloques de memoria que se almacenan en ella:

- **Directa**: en un conjunto solo cabe un bloque. Es decir, los conjuntos son de una sola vía. El campo «índice» indica el conjunto donde se almacena el bloque. Este tipo de organización lleva a un uso de comparadores para la etiqueta mínimo, ya que el bloque solo puede estar en un único lugar: en la única vía del conjunto indicado por el «índice». Sin embargo, causa más fallos debido a conflictos al acceder a bloques cuya ubicación coincide en el mismo conjunto.

- **Completamente asociativa**: La caché consta de un único conjunto, que almacena todos los bloques que hay en la caché y que tiene un número de vías igual al máximo número de bloques que puede almacenarse. Esto evita completamente los mencionados fallos por conflicto, pero requiere de un alto número de comparaciones con la etiqueta cuando se busca un bloque, ya que este puede encontrarse en cualquier vía del único conjunto de la caché. Esto lleva a un uso de área y consumo energético muy elevado debido a las comparaciones.

- **Asociativa por conjuntos**: Esta organización busca un equilibrio entre la política de ubicación «directa» y la «completamente asociativa». Como en la organización «directa», hay varios conjuntos que se seleccionan con el campo «índice», pero cada conjunto dispone de un número n de vías[1]. Dentro de cada conjunto, el bloque puede encontrarse en cualquiera de las vías de este, lo que limita el número de comparaciones a n cuando se busca un bloque.

Las organizaciones «completamente asociativa» y «asociativa por conjuntos» requieren un algoritmo de reemplazo que seleccione la vía a reemplazar en caso de que el conjunto destino se encuentre lleno y no haya espacio para almacenar un nuevo bloque. Existen varios tipos de estas políticas de reemplazo, destacando las siguientes:

- Reemplazo aleatorio: se selecciona una vía al azar, que será sustituida por la nueva.

- FIFO: la política «primero en entrar, primero en salir» (*first in, first out* en inglés, también conocida como *First come, first served* o simplemente «FCFS»), es aquella en la que la línea que reemplazamos del conjunto será la que lleve más tiempo en caché, es decir, la vía más antigua.

- LRU: la política LRU (que procede de las siglas en inglés de «*Least recently used*») funciona sustituyendo la vía a la que se lleva más tiempo sin acceder (la menos usada, de ahí su nombre) del conjunto.

[1] Es usual que a este tipo de organización se le conozca como «asociativa de n-vías».

2.1.3. Estructura y funcionamiento de un controlador de memoria

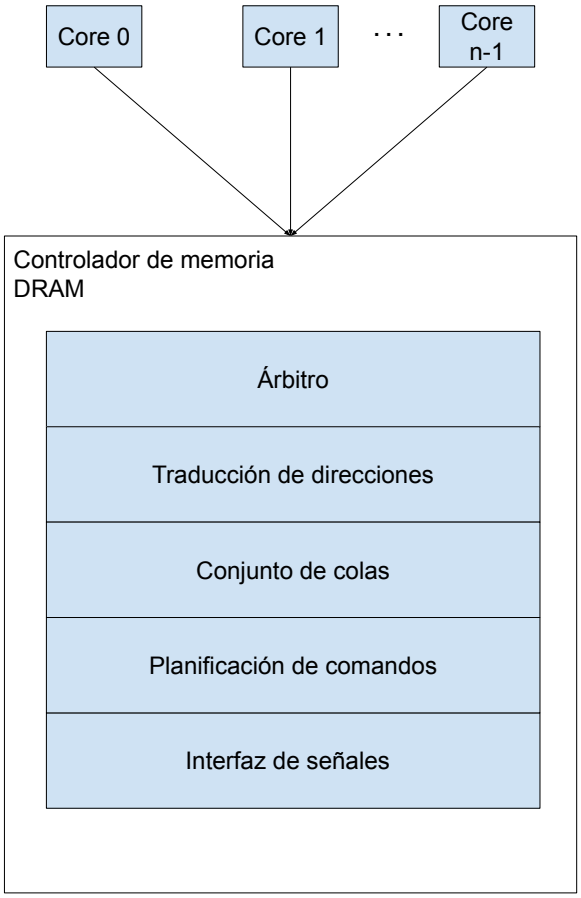

Figura 2.2: Estructura genérica de un controlador de memoria DRAM

En general, un controlador de memoria tiene la estructura mostrada en la figura 2.2. Como se puede observar, el controlador de memoria DRAM está dividido en cinco partes diferenciadas,[2] cada una de ellas con una funcionalidad específica dentro del controlador y conectadas de tal forma que la salida de una de estas partes está conectada a la entrada de la siguiente.

Cuando el procesador (uno o varios *cores*) ejecuta una instrucción de acceso a memoria, se busca el bloque referenciado en la jerarquía de caché. Si se llega al último nivel de caché y se produce un fallo, entonces se realiza un acceso a la memoria principal para traer el bloque. Este acceso debe pasar primero por el controlador de memoria, normalmente situado en el mismo chip del procesador, que se encarga de gestionarlo: almacena los accesos pendientes en colas, planifica el acceso y envía las señales oportunas a los módulos DIMM a través del bus de memoria. En otras palabras, el controlador de memoria

[2]Por partes nos referimos a funcionalidades diferentes que interactúan entre ellas para implementar el conjunto del controlador de memoria.

debe decidir qué petición quiere atender primero. Para ello, el controlador implementa un planificador o árbitro que decidirá a qué petición ofrecerá su servicio primero. Este puede ser un cuello de botella importante ya que es la única forma de «acceder» a memoria principal.

Cuando una petición gana el arbitraje y entra al controlador de memoria, se relaciona con una localización de las direcciones de memoria y tras ello, se convierte a una secuencia de comandos DRAM que se colocan en colas. Estas colas pueden ser comunes o pueden estar distribuidas de tal forma que exista una cola por rango o banco de memoria.

Tras almacenar los comandos en las colas del controlador de memoria, este selecciona los comandos a ejecutar según la lógica que utilice en la planificación de comandos. Existen diferentes factores que influyen en esta toma de decisión, como son, la prioridad de la petición, la disponibilidad de los recursos necesarios para servir la petición, la dirección del banco de memoria a la que hay que acceder o el histórico de accesos del agente que ha realizado la petición.

A continuación de la decisión de comandos a ejecutar, estos se ejecutan usando la interfaz de señales del controlador, que se encarga de realizar las señales eléctricas necesarias para ejecutar estos comandos. Estas ejecuciones tienen siempre en cuenta las parámetros temporales del DIMM DRAM con el que están conectados con tal de no hacer un uso incorrecto que pueda llevar a errores como datos inconsistentes.

2.1.4. Tecnología DRAM

Los módulos DIMM de memoria DRAM se estructuran internamente de manera jerárquica. En el nivel más pequeño se encuentran celdas DRAM que almacenan un único bit. Estas se agrupan en matrices con cierto número de filas y columnas. Para aumentar el paralelismo estas matrices se agrupan en bancos que pueden trabajar de manera independiente. El número de bancos depende de la tecnología de memoria. En DDR4 los módulos de memoria disponen de 16 bancos. Los datos se almacenan a lo largo de los chips que conforman el banco, por tanto, para acceder a un dato se precisa acceder a todos los chips del banco con las mismas órdenes. A su vez, múltiples chips con sus bancos se pueden agrupar, compartiendo bus de direcciones y órdenes pero contribuyendo diferente información, esta agrupación recibe el nombre de rango, o lo que es lo mismo, un DIMM completo. Múltiples DIMM juntos se utilizan para aumentar todavía más la capacidad de memoria principal, y se pueden interconectar de formas diferentes, dando por lo tanto latencias de acceso diferentes. Estas agrupaciones de varios DIMM son dirigidas por el controlador de memoria. Si añadimos varios controladores de memoria con diferentes rangos, tenemos la creación de varios canales de memoria independientes. [4, 5]

Como podemos observar en la figura 2.3, una celda de memoria DRAM está formada por un transistor, un condensador, y las líneas de fila (también conocidas como *word-line*) y de columna (también se conocen por *bit-line*). Para las operaciones de escritura, se fuerza la *bit-line* de todas las celdas que se quieren escribir al valor deseado. A continuación,

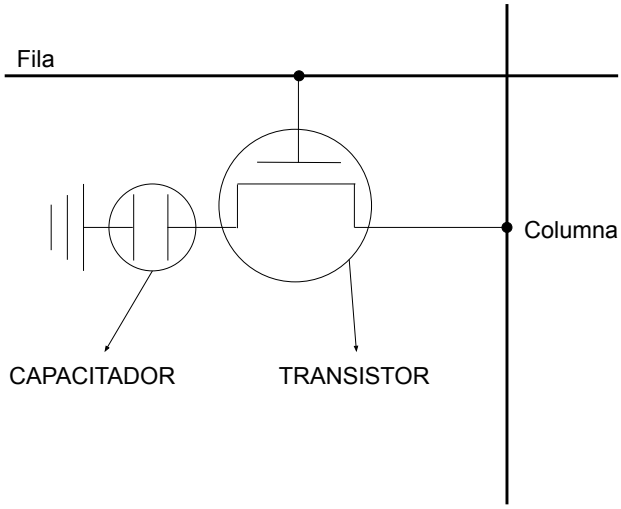

Figura 2.3: Celda de memoria DRAM.

se activa la fila de las celdas y de esta forma el condensador pasa a tener el uno o el cero lógico. Es importante mencionar que al ser condensadores los elementos eléctricos que almacenan el estado, tienen que ser refrescados frecuentemente para no perder el valor. Para las operaciones de lectura, las columnas se cargan a un valor intermedio entre el uno y el cero lógico, por ejemplo 0.5 V. Tras esto, se permite a la celda conducir corriente activando la fila, que supone una pequeña variación en el voltaje de la columna. Esta pequeña variación es aumentada por unos amplificadores (componente electrónico que aumenta el voltaje de una señal) que a su vez devuelven el valor guardado en la fila.

2.1.5. Tecnología NV

En contraposición a la celda DRAM, la arquitectura de la celda de memoria para el DIMM Intel© Optane™ está formado por la tecnología 3D XPoint™. Como se puede ver en la figura 2.4, una celda está formada por un selector (bloque amarillo) y un elemento de memoria (bloque verde). Este par de elementos se rodea de dos conductores (paralelepípedos rectangulares en color gris oscuro) por el lado superior e inferior, que sirven para direccionar de forma única a una sola celda. Para modificar la información, se aplica un voltaje a los conductores, la intensidad de este voltaje determina la escritura de un cero o un uno lógico. De esta forma, el selector modifica las propiedades físicas del material del que están formados los elementos de memoria (que suele ser Telurio, Selenio, Antimonio, Níquel o Germanio), cambiando su resistividad y el valor lógico que almacena [6, 7].

Una clara ventaja que tiene la tecnología 3D XPoint™ frente a DRAM es como sus celdas pueden agruparse unas encima de otras, siendo este el motivo de su mayor densidad de almacenamiento. Esta es la característica que aprovecha Intel© con tal de obtener las memorias OPTANE™.

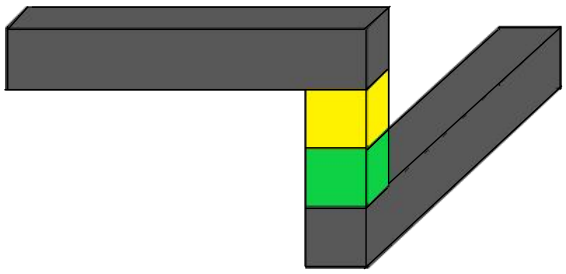

Figura 2.4: Celda de memoria usada en las memorias OPTANE.

Estudios previos

Debido a la evolución continua de los sistemas: procesadores más potentes, procesadores con más núcleos, más jerarquía de cache on-chip... Siempre es necesario continuar con el esfuerzo para mejorar las prestaciones del subsistema de memoria y evitar que este limite las prestaciones del sistema.

En este sentido se ha realizado mucho esfuerzo tanto a nivel de TFG como de TFM y de tesis doctoral. Respecto a los TFGs defendidos, el objetivo de estudio varía ampliamente en ellos. Desde buscar un aumento del ancho de banda mediante una planificación más avanzada [8], pasando por la simulación de nuevas arquitecturas de memorias caché para sistemas empotrados [9] hasta el diseño de cachés de primer nivel con nuevas tecnologías [10]. Algunos de estos TFGs mencionados, en concreto el primero y el último, continuaron evolucionando con el TFM y posteriormente la realización de la tesis doctoral.

Respecto a los TFGs mencionados, no hemos encontrado ningún otro trabajo que trate de implementar una caché en el controlador de memoria con tal de obtener una mejora de prestaciones. Por el potencial que este cambio tiene para acelerar la jerarquía de memoria, nos hemos decidido por realizar este TFG.

2.2 Simuladores del estado del arte

2.2.1. Simuladores de memoria NVMM

Al ser la NVMM/NVRAM una tecnología novedosa actuando como memoria principal, no hay muchos simuladores que las soporten. Por ejemplo, DRAMsim3 [11] o ramu-

lator [12], simuladores de memoria principal ampliamente utilizados por la comunidad científica, no soportan este tipo de tecnologías. Es por ello que hemos buscado otras opciones que nos permitan simular el funcionamiento de esta tecnología de forma precisa, además de poder ejecutarse de forma conjunta con otros simuladores de CPU que modelen el núcleo computacional como pueden ser Gem5 [13] o ZSim [14]. Los simuladores de memoria principal no volátil que hemos estudiado son NMTsim [15], VANS [16] y NVMain [17].

NMTsim

Es un simulador reciente que implementa el protocolo NVDIMM-P [18], pensado para poder tener diferentes módulos DIMM de diferentes tecnologías en un mismo canal de memoria mediante el uso de peticiones a los rangos de forma asíncronas y fuera de orden.

El simulador implementa dos controladores distintos en el «controlador de memoria»: uno para módulos DRAM y otro con NVDIMM-P. Ambos controladores coexisten en el mismo canal pudiendo ofrecer servicio así tanto a DRAM como a NVRAM.

Además de implementar un estándar tan reciente, también intenta añadir un modelo fiable de parámetros temporales de las memorias NVRAM. Usualmente, esto se realiza mediante la aproximación de las características de las memorias NVRAM a memorias DRAM, generalmente añadiendo ciclos de latencia al modelo de un DIMM DRAM para implementar el retardo adicional. Esto no es preciso, ya que las memorias NVRAM difieren en su funcionamiento de las DRAM. Entre otras diferencias, las memorias no volátiles no requieren de la orden «REFRESH», necesaria para el correcto funcionamiento de las DRAM. El protocolo NVDIMM-P también añade otros comandos propios. NMTsim permite la ejecución con Gem5, por lo que sería un candidato para este proyecto. Lamentablemente, a fecha de la creación de esta memoria no está disponible al público.

VANS

Validated cycle-Accurate NVRAM Simulator [16] (a partir de ahora VANS) es un simulador NVRAM DIMM que modela la arquitectura de Optane™ DIMM. Este simulador también se puede conectar con Gem5 o puede ser usado en solitario con trazas de acceso a memoria principal.

Aunque en principio modela solo la arquitectura del DIMM Optane™, podemos modificar algunos parámetros para simular otros tipos de arquitecturas NVRAM. Para ello, hay que usar LENS (*Low-level profilEr for Non-volatile memory Systems*) [16], una herramienta que obtiene los parámetros que necesita VANS para modelar una arquitectura de memoria principal no volátil. LENS obtiene estos parámetros mediante el uso de tres *probers*, que son pequeños *benchmarks* encargados de encontrar estos parámetros a través de un análisis de las estadísticas de acceso a memoria proporcionadas por el sistema operativo.

Aunque VANS es una gran herramienta para simular con detalle NVRAM, no ofrece una forma sencilla de modificar el subsistema de memoria que simula. Es por ello que no nos sirve para realizar las modificaciones que queremos llevar a cabo en este TFG.

NVMain

NVMain es un simulador preciso a nivel de ciclo de memoria principal para memorias no volátiles y DRAM. Al igual que el resto, lo podemos usar junto con otros simuladores como Gem5 o ZSim[3] para simular un sistema completo. También se puede usar de forma *standalone* mediante el uso de trazas de memoria para simular las peticiones a memoria principal a partir de estos ficheros.

Este simulador permite la implementación de componentes del sistema de memoria principal de forma sencilla, mediante la extensión de clases base definidas en C++ y de ejemplos con los que viene incluido. También da soporte para simular memoria DRAM convencional y sistemas de memoria híbridos, donde DRAM y NVRAM trabajan de forma combinada, eso si, en diferentes canalaes. En este TFG esta característica no se va a usar, pero en trabajos futuros será clave.

Además, se puede modelar la arquitectura de los rangos de memoria mediante los archivos de configuración, como pueden ser parámetros de consumo de energía y parámetros temporales relevantes al tipo de memoria. Esto hace de NVMain un simulador muy flexible, ideal para modificar la interconexión del sistema de memoria o controladores de memoria, como es el caso en este TFG.

2.2.2. Simuladores de procesador

El diseño de hardware requiere de herramientas de simulación para poder implementar y evaluar las modificaciones realizadas en la arquitectura. Se trata de herramientas software complejas que modelan con detalle ciclo-a-ciclo los distintos eventos que acontecen en la microarquitectura del procesador. En las dos últimas décadas, tanto para el diseño de procesadores con un solo núcleo como multinúcelo, se han utilizado distintos entornos de simulación que modelan la microarquitectura de estos procesadores avanzados.

En este apartado se presentan algunos de los simuladores de arquitectura más conocidos y utilizados por la comunidad científica, identificando el que se elegirá para este proyecto.

Multi2Sim

Multi2Sim [19] es un simulador de microarquitectura detallada ciclo-a-ciclo que modela la ejecución de programas para el conjunto de instrucciones -o ISA- MIPS32. El simulador implementa de manera detallada los componentes internos de cada una de las etapas del *pipeline* del procesador. En este esquema, un módulo temporal se encarga de

[3] El funcionamiento con ZSim no está soportado de forma base, es un proyecto de terceros.

revisar las eventos que acontecen en distintas etapas por orden inverso a como se encuentran implementadas en el *pipeline*. Tras esto, un módulo funcional se encarga de simular la ejecución de las instrucciones. Gracias a esta decisión de diseño, Multi2Sim implementa una simulación detallada usando la perspectiva *timing-first*, donde se analizan las instrucciones máquina ejecutadas recientemente y se derivan las latencias operacionales a partir de esta información. El simulador soporta procesadores multihilo simultáneo y multinúcleo.

ZSim

ZSim es un simulador de la plataforma x86-64, capaz de modelar y simular sistemas con ejecución «en orden» como «fuera de orden», con jerarquías de memoria configurables e incluyendo modelos temporales complejos. Pese a ser capaz de simular sistemas heterogéneos con un elevando número de núcleos, consigue velocidades de simulación muy elevadas gracias al uso que realiza de los procesadores lógicos (o *threads*) en los procesadores actuales.

Este simulador posee unas grandes características para ser el simulador complementario a NVMain. Lamentablemente, NVMain no dispone de soporte oficial de ZSim, ya que este soporte es un proyecto de terceros.

Gem5

El simulador Gem5, es un simulador modular que funciona mediante eventos. Gem5 ofrece una gran flexibilidad a la hora de elegir el sistema a simular. Esto se consigue mediante el uso de complejos ficheros de configuración (*scripts* programados en Python que inicializan los componentes del sistema simulador y lanzan la simulación). Este simulador permite configurar no solo el tipo de CPU o de memoria principal a usar, sino también realizar configuraciones más específicas, como por ejemplo determinar las conexiones entre cachés y la CPU o memoria principal. Gem5 ofrece dos tipos distintos de simulación, el modo emulación de llamadas al sistema[4] (*syscall emulation* en inglés, también conocido como SE) y el modo de simulación de sistema completo (conocido como *full-system* en inglés o FS).

En modo FS, el simulador emula un sistema entero desde el arranque, como si se tratase de una máquina real. Por ello hay que aportarle una imagen de disco con un sistema operativo (Ubuntu 18.04 en nuestro caso) y un *kernel*. De esta forma se consigue una simulación mucho más real que en modo SE. Lamentablemente, para conseguir este nivel de precisión se incrementa el tiempo de simulación, ya que la simulación de sistema completo conlleva la simulación del arranque del sistema.

Gem5 ofrece ejemplos ya creados de archivos de configuración, como **fs.py** y **se.py** correspondientes a los modos FS y SE respectivamente. Estos ficheros de configuración permiten parametrizar, desde la linea de órdenes, gran parte de la arquitectura del siste-

[4]Las llamadas al sistema son un conjunto de funciones que ofrece el sistema operativo a los procesos en ejecución.

ma a simular. Existen gran cantidad de parámetros, destacando los siguientes, que son los utilizados en este TFG para los *scripts* de lanzamiento de las simulaciones:

- –cpu-type. Permite elegir qué tipo de CPU queremos usar para la simulación. Puede ser algunos de los que Gem5 ya define (como «TimingSimpleCPU» o «AtomicSimpleCPU») o también se puede definir uno propio y usarlo.
- –caches. Especifica que queremos utilizar cachés de nivel 1.
- –l1i_size. Define el tamaño (en bytes) de la caché de nivel 1 de instrucciones.
- –l1i_assoc. Establece la asociatividad (número de vías) que cada conjunto de caché puede almacenar.
- –l1d_size y –l1d_assoc. Análogos a «l1i_size» y «l1i_assoc» pero para la caché de nivel 1 de datos.
- –l2cache. Establece que queremos utilizar una caché de nivel 2.
- –l2_size. Define el tamaño (en bytes) de la caché de nivel 2.
- –l2_assoc. Define la asociatividad de la caché de nivel 2.
- –kernel (solo en modo FS). Indica el path donde se encuentra el fichero kernel compilado.
- –disk-image (solo en modo FS). Indica el path de la imagen de disco de tipo **.img**.
- –script (solo en modo FS). Indica el path en el sistema de archivos de la imagen de disco con el fichero de órdenes a ejecutar una vez el sistema operativo haya arrancado por completo y se puedan ejecutar programas.

2.2.3. Simuladores elegidos para la realización del TFG

Tras el estudio de los distintos simuladores, hemos optado por utilizar NVMain como simulador de memoria no volátil gracias a la facilidad que ofrece para modificar los elementos del subsistema de memoria. Junto a este, usaremos Gem5, un simulador muy potente y flexible con el que NVMain se puede compilar de forma conjunta.

CAPÍTULO 3

Entorno experimental

Para lanzar los experimentos hemos utilizado un sistema como el que se muestra en la tabla 3.1. Para los pasos que comentamos a continuación es crítico el uso del sistema operativo «Ubuntu 18.04.5». Esto se debe a que algunos de los paquetes que son prerrequisito de alguno de los simuladores utilizados no están disponibles, o al menos no su versión correcta, en otras distribuciones de Linux.

CPU	Intel© i5-11600 @ 2.8 GHz
Núcleos (Hilos)	6 (12)
Caché L1 Datos (por núcleo)	48 KB
Caché L1 Instr. (por núcleo)	32 KB
Caché L2	3 MB
Caché L3	12 MB
Memoria Principal	1x32 GB DDR4-2400 DRAM
Sistema Operativo	Ubuntu 18.04.5
Kernel	5.10.0-1026-oem

Tabla 3.1: Características del sistema utilizado para el lanzamiento de simulaciones.

A continuación se describe la configuración de los simuladores utilizada en los experimentos realizados en este TFG y los *benchmarks* utilizados.

3.1 Instalación y configuración del simulador Gem5

En este apartado explicamos el aspecto práctico para la obtención y compilación de Gem5. Por ello ofrecemos una guía con los pasos y las ordenes necesarias con tal de conseguir el binario de este. En un principio este no dispondrá de NVMain, pero la compilación cruzada se realizará en el apartado 3.3.

3.1.1. Preparación del entorno para Gem5

Lo primero que necesitamos son los prerrequisitos de Gem5. En la documentación que Gem5 nos ofrece, existe una orden para instalarlas dependiendo de la versión de Li-

nux que tengamos instalada. A esta hemos añadido los requisitos de NVMain, de esta forma unificamos ambas instalaciones en una única orden. Para obtener todas las dependencia de Ubuntu 18.04 ejecutamos la siguiente orden:

```
    sudo apt install build-essential git m4 zlib1g zlib1g-dev libprotobuf-dev
protobuf-compiler  libprotoc-dev libgoogle-perftools-dev python-dev
libboost-all-dev gcc-7 g++-7 curl libpng-dev
```

Para facilitar la futura compilación con NVMain, recomendamos también instalar (o hacer *downgrade* si ya están instalados en una versión diferente) gcc-7 y g++7, y actualizar las alternativas por defecto con la siguiente orden:

```
    sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 70
--slave /usr/bin/g++ g++ /usr/bin/g++-7 --slave /usr/bin/gcov gcov
/usr/bin/gcov-7
```

Para la instalación de SCons es necesario obtener la versión 2.5.1, por ello, vamos a obtenerlo mediante la orden wget, a continuación procederemos a su instalación como se muestra con las siguientes ordenes:

```
    wget https://sourceforge.net/projects/scons/files/scons/2.5.1/
scons-2.5.1.tar.gz/download
    tar -xzvf scons-2.5.1.tar.gz
    cd scons-2.5.1
    python setup.py install
```

Una vez concluida la instalación, podemos eliminar el directorio que se ha creado al descomprimir el fichero con los ficheros de instalación, esto se puede llevar a cabo con las siguientes ordenes:

```
    cd ..  #Esta solo hay que ejecutarla si seguimos en el directorio
    rm -r scons-2.5.1
```

Una vez hemos realizado la instalación de SCons, podemos proceder a clonar el repositorio de Gem5 versión 20.0.0.3 desde el servidor donde está alojado. Recomendamos crear un directorio para guardar los repositorios tanto de Gem5 como de NVMain y de esta forma tener ambos simuladores en un directorio de trabajo padre. Si queremos crear este directorio, podemos usar la orden:

```
    mkdir gem5-nvmain
```

Una vez creado el directorio, nos movemos a él con la orden:

```
    cd gem5-nvmain
```

Entonces podemos clonar el repositorio con la orden git tal que:

```
git clone https://gem5.googlesource.com/public/gem5 --branch v20.0.0.3
```

Tras obtener los ficheros necesarios desde el repositorio, debemos instalar «pip», y usarlo para instalar el paquete «six». Es importante que se instale para Python2. Esto se puede conseguir con el siguiente conjunto de órdenes:

```
cd gem5
curl https://bootstrap.pypa.io/pip/2.7/get-pip.py --output get-pip.py
sudo python2 get-pip.py
pip install six
rm get-pip.py
```

Al acabar de obtener el paquete six en la carpeta de Gem5 podremos realizar la compilación del simulador. Para ello usaremos SCons, que instalamos anteriormente y es un programa similar al conocido Make, pero estando este escrito en el lenguaje de programación Python. Para ello debemos ubicarnos en el directorio base de Gem5 (la carpeta que ha creado git al obtener la copia del repositorio) y ejecutar:

```
scons build/X86/gem5.opt -j $(nproc)
```

Si al momento de compilar por primera vez nos muestra un mensaje avisando de que falta «gem5 style» o «commit message hook», simplemente tenemos que presionar la tecla intro, esto es normal. Indicar que no tiene que ser necesariamente el binario «gem5.opt» el objetivo de la compilación, existen diferentes tipos como «gem5.debug», el propio «gem5.opt» o «gem5.fast», que puede acelerar algunas simulaciones. Finalmente, el parámetro «-j $(nproc)» especifica la cantidad de *threads* a usar en el momento de la compilación. Recomendamos usar la máxima cantidad posible de hilos ya que la compilación se acelera mucho gracias a esto, en caso de querer dedicar una cantidad menor que todos los hilos disponibles en el procesador, se puede sustituir «$(nproc)» por la cantidad a usar.

3.1.2. Lanzamiento de simulaciones SE

Tras conseguir compilar el binario podremos realizar la primera simulación. Esta será del tipo *system call emulation*. Para ello simplemente llamamos al binario que acabamos de compilar y le pasamos un archivo de configuración encargado de darle estructura al sistema donde se ejecutará el programa. Esto se realiza con la siguiente orden:

```
build/X86/gem5.opt configs/example/se.py --cpu-type=TimingSimpleCPU
--caches --l1i_size=32kB --l1d_size=32kB --l2cache --l2_size=2MB
--mem-type=DDR4_2400_4x16 -c tests/test-progs/hello/bin/x86/linux/hello
```

Esta simulación se encargará de ejecutar un programa que mostrará «Hello World!» por consola. Es importante tener en cuenta que si queremos usar binarios propios, estos deben estar compilados de forma estática (con las librerías incluidas en el binario) ya que no se puede obtener acceso a las librerías dinámicas que suele ofrecer linux.

3.1.3. Lanzamiento de simulaciones FS

Para poder ejecutar simulaciones de sistema completo es necesario obtener un fichero de imagen de disco y un kernel compilado. Lo más sencillo es obtener una imagen y un kernel ya preparados de la página web de recursos de Gem5 [20].

También se pueden crear manualmente, pero en todas las pruebas que hemos realizado no ha sido posible hacer boot de forma correcta. A la hora de hacer el arranque ocurría un «kernel panic error» deteniendo la simulación. En nuestro caso usamos la imagen parsec.img y el kernel versión 4.19.83, ambos disponibles en la página web de recursos de Gem5. Es recomendable agrupar estos recursos en una carpeta, con dos subdirectorios, un directorio «disks», donde se dejarán las imágenes de disco, y otro directorio denominado «binaries», donde irán los *kernels* compilados.

Tras hacer esto nos interesa crear una variable de entorno denominada «M5_PATH». De esta forma no hará falta indicar todo el path hasta la imagen de disco ni el kernel, solo desde esta carpeta base hasta el directorio correspondiente según el tipo y el nombre del fichero. Después de conseguir tanto la imagen de disco como el fichero con el kernel en el directorio al que apunta la variable de entorno definida, podemos ejecutar una simulación de sistema completo. Para ello, y de igual forma muy similar que para la simulación SE, nos situamos en la carpeta base de Gem5 y ejecutamos la siguiente orden:

```
build/X86/gem5.opt configs/example/fs.py --mem-type=DDR4_2400_4x16
--cpu-type=AtomicSimpleCPU --kernel=$M5_PATH/binaries/vmlinux-4.19.83
--disk-image=$M5_PATH/disks/parsec.img --caches --l1i_size=32kB --l1i_assoc=8
--l1d_size=32kB --l1d_assoc=8 --l2cache --l2_size=4096kB --l2_assoc=8
```

Veremos como la terminal empieza a escribir mensajes de Gem5, entre ellos, habrá uno que avisará de un puerto donde nos podemos conectar. Este puerto suele ser el 3456 a no ser que esté ocupado, en cuyo caso se prueba con el siguiente hasta encontrar un puerto libre. Llegará un momento que Gem5 nos avisará de que la simulación ha comenzado. Cuando esto ocurra, nos podemos conectar a la terminal de este sistema simulado usando la orden (suponiendo que ha avisado de que la conexión se puede realizar al puerto 3456):

```
telnet localhost 3456
```

Es importante comentar que si solo nos interesa ver la salida a consola de este sistema podremos observarla en el fichero system.pc.com_1.device del directorio «m5out». La conexión mediante telnet nos sirve de consola interactiva. Si en algún momento queremos

salir de la sesión de telnet, se puede realizar mediante la combinación de teclas «ctrl+5» y seguido teclear «close» y terminar presionando «intro».

Una vez se ha arrancado por completo y tengamos el *prompt* de Ubuntu 18.04, el sistema operativo instalando en la imagen de disco parsec.img, podemos interactuar con él como si se tratase de un sistema normal aunque mucho más lento y con funcionalidades restringidas. Si queremos detener la simulación ejecutaremos la siguiente orden en el sistema simulado si tenemos control de este:

```
/sbin/m5 exit
```

En caso de no tener el control mediante telnet, podemos usar el atajo de teclado «ctrl+C» en la terminal que está ejecutando la simulación para concluir con esta. El binario /sbin/m5 está creado especialmente por Gem5 para poder interactuar con el sistema «*host*» desde el sistema simulado, desde la creación de *checkpoints* hasta la lectura de ficheros del sistema host al sistema simulado.

3.1.4. Detalles de la configuración para las simulaciones

La configuración que usamos con Gem5 es la siguiente:

Elemento	Valor
CPU	TimingSimpleCPU
Tipo de memoria	NVMainMemory[1]
Archivo de configuración de NVMain	Varía según el tipo de memoria que queramos simular[2]
Kernel	4.19.83
Imagen de disco	parsec.img
Caché datos L1	32 KB. Asociatividad de 8 vías.
Caché instrucciones L1	32 KB. Asociatividad de 8 vías
Caché instrucciones L2	256 KB. Asociatividad de 8 vías

Tabla 3.2: Configuración de Gem5 usada en las simulaciones para este TFG

[1] Explicamos el valor de este campo más adelante.
[2] Explicamos el valor de este campo más adelante.

3.2 Instalación y configuración del simulador NVMain

De forma similar a como se ha hecho con Gem5; a continuación, se exponen los pasos necesarios para la obtención del ejecutable de NVmain de manera independiente a Gem5, también conocido como ejecución *standalone*.

3.2.1. Preparación del entorno para NVMain

NVMain tiene escasas dependencias y todas se comparten con Gem5, por ello, las instalamos de forma conjunta en la sección 3.1.1. En cualquier caso, conviene recordar las versiones necesarias para el funcionamiento de este.

- GCC y G++: versión 7.
- Python: versión 2.7.17 o 2.7.18, es probable que funcione con versiones más antiguas pero estas dos son las que han sido probadas a lo largo de la realización de este TFG.
- SCons: versión 2.5.1.

Para obtener los ficheros fuente de NVMain debemos clonar el repositorio que contiene uno de los parches necesarios preaplicado para la compilación cruzada. Este no tendrá efecto en la configuración *standalone*. Para obtener este repositorio ejecutamos la siguiente orden:

```
git clone --branch co_gem5 https://github.com/OnceMore2020/NVmain.git
```

Recomendamos que esta orden se ejecute en el directorio padre de trabajo que hemos creado en la fase de instalación de Gem5. Una vez tengamos el repositorio clonado en nuestra máquina, podemos comenzar la compilación *standalone*, es decir, sin Gem5 (esta se discute en el próximo apartado). Para ello, ejecutamos la siguiente orden en el directorio base de NVMain:

```
scons --build-type=fast
```

De modo similar a Gem5, NVMain dispone de diferentes binarios posibles a la hora de compilar. En este caso son *fast*, *debug* o *prof*. Vamos a generar el binario *fast* para usarlo como prueba de las dependencias.

Al compilar, se producirá un error debido a que estamos intentando usar clases de Gem5 («NVmain/SConscript» línea 36). Como estamos realizando la compilación en solitario –o *standalone*– podemos solucionar este error simplemente comentando esta línea con una almohadilla «#». Una vez comentada, la compilación del binario de NVMain funcionará sin problemas.

3.2.2. Lanzamiento de simulaciones standalone en NVMain

Para poder lanzar simulaciones *standalone* de NVMain se requieren el binario de NVMain compilado y tres parámetros. El primero es un fichero de configuración de NVMain; afortunadamente, el repositorio nos ofrece varios ya creados en el directorio «config» dentro del directorio base de NVMain. El segundo, necesitamos un archivo de trazas; de nuevo, el repositorio proporciona trazas para poder realizar las pruebas iniciales. Finalmente, debemos decidir cuántos ciclos queremos ejecutar en la simulación. A continuación se presenta una posible orden para lanzar una simulación de prueba, ejecutada desde el directorio base de NVMain:

```
./nvmain.fast Config/2D_DRAM_example.config Tests/Traces/hello_world.nvt
100000
```

Esta simulación debería tardar poco tiempo y, al acabar, NVMain nos devolverá una gran variedad de estadísticas sobre esta por consola. Si queremos almacenar la salida de NVMain en un fichero podemos usar una redirección a un fichero usando el operador «>» seguido del nombre del fichero que se creará con la salida. Por ejemplo:

```
./nvmain.fast Config/2D_DRAM_example.config Tests/Traces/hello_world.nvt
100000 > fichero_salida_nvmain.txt
```

3.3 Compilación cruzada Gem5-NVMain

Una vez tenemos tanto el ejecutable de Gem5 como el de NVMain y hemos comprobado el funcionamiento de ambos simuladores por separado, podemos proceder a la compilación cruzada. Este es el paso más importante ya que su resultado nos otorga el simulador que implementa NVMain con Gem5 y el que usaremos para realizar las pruebas y obtener resultados de las modificaciones que hagamos.

Antes de nada, debemos descomentar la línea 36 del fichero «SConscript» para la compilación. Además en el fichero «Options.py», ubicado en el directorio «configs/common/» accesible desde el directorio base de Gem5 y Justo tras la definición de la función «addNoISAOptions()», entre las líneas 86 y 87[3] añadiremos el siguiente código respetando las tabulaciones de Python y dejando una línea en blanco tras realizar el cambio:

```
for arg in sys.argv:
    if arg[:9] == "--nvmain-":
        parser.add_option(arg, type="string", default="NULL",
            help="Set NVMain configuration value for a parameter")
```

Una vez hemos modificado este archivo, y suponiendo que el resto de pasos se han llevado a cabo de forma correcta, podremos compilar Gem5 avisando a SCons de que añada a NVMain a la compilación. Esto lo podemos conseguir con la siguiente orden:

[3]Cada vez que nos referimos a una línea o conjunto de ellas, es tras las modificaciones anteriores ya realizadas. Esto se hace así para facilitar la implementación del código.

```
scons build/X86/gem5.opt EXTRAS=../NVmain -j $(nproc)
```

En caso de que no hayamos podido realizar alguno de los pasos necesarios hasta llegar a este punto, se ha publicado un repositorio en github[4] con todos los ficheros necesarios y preparados para la compilación. El argumento «EXTRAS» de SCons solo necesita el path hasta el directorio base de NVMain. Una vez se haya compilado el binario de Gem5-NVMain, podemos lanzar simulaciones modificando algunos argumentos de la orden de Gem5 para que este llame a NVMain y actue como su sistema de memoria, estas modificaciones son:

- cpu-type: esto se debe a que NVMain solo funciona correctamente cuando Gem5 envía paquetes de tipo *timing*. No vamos a entrar en detalle sobre los modos internos de paquetes de Gem5, solo indicar que la CPU debe enviar paquetes en el modo *timing*: una CPU que viene ya definida con Gem5 capaz de ello es TimingSimpleCPU.

- mem-type: aquí debemos indicar a Gem5 que queremos que NVMain sea su modelo de memoria, es por ello que el valor de este parámetro debe ser «NVMainMemory».

- nvmain-config: este parámetro es uno nuevo que el cambio que aplicamos a Gem5 generó. Como NVMain necesita un fichero de configuración para poder funcionar, este parámetro se encarga de ofrecérselo para que pueda crear el sistema de memoria. El valor debe ser el path hasta un fichero de configuración de NVMain.

Por ejemplo, una orden para poder usar el simulador híbrido Gem5-NVMain podría ser:

```
build/X86/gem5.opt configs/example/fs.py --cpu-type=TimingSimpleCPU
--caches --l1i_size=32kB --l1i_assoc=8 --l1d_size=32kB --l1d_assoc=8
--l2cache --l2_size=256kB--l2_assoc=8 --mem-type=NVMainMemory
--disk-image=$M5_PATH/disks/parsec.img
--kernel=$M5_PATH/binaries/x86_64-vmlinux-4.19.83
--nvmain-config=../NVmain/Config/2D_DRAM_example.config
```

3.4 Cargas de ejecución

Para la evaluación de resultados, hemos ejecutado diferentes *benchmarks*. Mediante la ejecución de estos en el binario cruzado, se han obtenido distintas trazas. En este apartado se presentan los que hemos usado, se explica como instalarlos en el sistema simulado, y finalmente se describe la información recabada en las trazas.

[4] https://github.com/miavgu/gem5-nvmain-tfg Último acceso 24 de agosto de 2021.

3.4.1. Benchmarks

Hemos usado distintos *benchmarks*, representativos de cargas reales de servidores, para estudiar el comportamiento del sistema y poder medir prestaciones. Las aplicaciones que usaremos como *benchmarks* son Redis y MySQL.

Redis es una estructura de almacenamiento en memoria que puede usarse como base de datos, caché y broker de mensaje. Redis ofrece diferentes estructuras de datos para que se puedan almacenar como *hashes*, listas, conjuntos, conjuntos ordenados y demás. Este *software* dispone de replicación de forma nativa, política de reemplazo LRU y diferentes niveles de persistencia en disco. Este servicio dispone de un programa separado no obligatorio conocido como «redis-benchmark», que se ocupa de cargar el servidor a base de peticiones con tal de ver el rendimiento en el sistema donde se ha desplegado. Este *benchmark* permite así comparar prestaciones entre sistemas, gracias al uso de ciertas operaciones que ofrece Redis, como pueden ser llamadas al servicio de tipo «get», «set» y demás.

MySQL es uno de los sistemas gestores de bases de datos más conocido y usados actualmente. Es una aplicación de código abierto y ofrece una gran variedad de características clave como son recuperación en caso de errores y soporte para operaciones *rollback* y *commit* entre otras. MySQL ofrece servicio a grandes empresas como Facebook, Google o Twitter. Para la obtención de estadísticas usamos SysBench, pensado para realizar pruebas sobre el sistema, nos permite realizar pruebas a sistemas gestores de bases de datos aunque también se puede usar para realizar pruebas arbitrariamente complejas a un sistema sin necesidad de que exista una base de datos.

3.4.2. Instalación de los benchmarks en la imagen de disco

Como se ha comentado en la sección 3.1.3, para realizar las simulaciones FS de Gem5 necesitamos tener una imagen de disco que represente un disco duro para el sistema simulado. Este disco no se puede modificar desde las simulaciones de Gem5 y es por ello que, si queremos añadir cualquier tipo de *software* a esta imagen de disco, debemos montarlo sobre un sistema ya existente para hacerlo. En este apartado se mostrará como podemos montar la imagen de disco y cómo añadir los *benchmarks* a esta imagen.

Montaje de la imagen de disco

Para montar la imagen de disco en un sistema linux debemos disponer de permiso para ejecutar la orden «sudo», ya que la orden mount es de uso exclusivo para superusuarios. Empezaremos montando la imagen de disco al sistema de archivos, para ello desde el directorio que contiene el fichero «parsec.img», ejecutaremos la siguiente orden:

```
sudo /bin/mount -o loop,rw,offset=1048576 parsec.img /mnt
```

Esta orden monta la imagen de disco en el directorio «/mnt». Tras esto, vamos a enlazar los directorios del sistema «/sys», «/dev» y «/proc» a los de la imagen montada, para ello ejecutamos estas ordenes:

```
sudo /bin/mount -o bind /sys /mnt/sys
sudo /bin/mount -o bind /dev /mnt/dev
sudo /bin/mount -o bind /proc /mnt/proc
```

Tras enlazar estos directorios, podemos ejecutar una terminal en el sistema con el disco imagen usando la orden:

```
sudo /usr/sbin/chroot /mnt /bin/bash
```

Esta consola nos permitirá usar la imagen de disco como si fuese el sistema por defecto, consiguiendo así poder instalar, eliminar y gestionar el sistema de forma convencional. Indicar que si necesitamos instalar programas desde ficheros fuente, hay que moverlos a «/mnt» con antelación, ya que una vez hemos abierto esta terminal no podremos acceder al sistema host.

Desmontaje de la imagen de disco

Desmontar la imagen de disco de forma correcta sirve para guardar los cambios en la imagen de vuelta al archivo. Para ello debemos de ubicarnos en cualquier directorio diferente a «/mnt», por ejemplo en nuestro *home* con la orden:

```
cd
```

A continuación, podemos desvincular los directorios del sistema host de los de la imagen de disco, para ello ejecutamos:

```
sudo /bin/umount /mnt/sys
sudo /bin/umount /mnt/proc
sudo /bin/umount /mnt/dev
```

Una vez desmontadas estas, podemos desmontar la imagen de «/mnt» usando la orden:

```
sudo /bin/umount /mnt
```

Instalación de Redis y Redis-Benchmark

Para la instalación de Redis vamos a suponer que ya está la imagen montada. La instalación de Redis se realiza desde el código fuente. Para obtener estos ficheros tenemos que ejecutar la orden:

```
wget http://download.redis.io/redis-stable.tar.gz
```

Una vez tenemos los fuentes, podemos descomprimir y mover los ficheros a la imagen de disco.

```
tar xvzf redis-stable.tar.gz
sudo mv redis-stable /mnt
```

Tras mover los ficheros fuente a la imagen de disco, realizamos el comando chroot y los compilamos con las siguientes ordenes:

```
sudo /usr/sbin/chroot /mnt /bin/bash
cd redis-stable
make
```

A continuación, podemos mover los binarios que hemos obtenido como resultado de la compilación a un directorio en la variable «PATH», una forma de realizar esto puede ser con la orden:

```
cp src/redis-server /usr/local/bin/
cp src/redis-cli /usr/local/bin/
cp src/redis-benchmark /usr/local/bin/
```

Al acabar la copia de los archivos, la imagen de disco estará lista para ser desmontada y ejecutar «redis-benchmark». De cara al lanzamiento de los *benchmarks* de Redis en la simulación, es importante anotar que hay que ejecutar primero el servidor Redis con la orden:

```
redis-server &
```

Antes de ejecutar el programa encargado de lanzar las pruebas al servidor con la orden:

```
redis-benchmark -n 1000 -t get,set,lpush,lpop
```

«redis-benchmark» usa dos parámetros; uno para indicar qué pruebas se ejecutan (-t) y otro que indica cuántas peticiones por prueba (-n). Nótese que antes de ejecutar esta orden es necesario esperar algo de tiempo a que el servidor Redis acabe de iniciarse. Una posible opción es utilizar la orden *sleep*. Un ejemplo de *script* utilizado en la simulación se puede ver en el anexo A.

Instalación de MySQL y SysBench

MySQL y SysBench son los últimos programas instalados para la utilización de estos *benchmarks*. La instalación de estos es más sencilla ya que es suficiente ejecutar el comando:

```
sudo apt-get install sysbench mysql-server -y
```

Tras instalar estos paquetes, vamos a crear un fichero en «/root» para que MySQL pueda leer las credenciales desde este, facilitando así las ejecuciones de este *benchmark*. Este fichero debe llamarse «.my.cnf» y contendrá las siguientes líneas:

```
[mysql]
user=root
password=12345
```

Esto sería todo lo necesario en lo referente a la instalación. Para el lanzamiento de los *benchmarks* se tienen tres pasos diferenciados. Primero, la preparación del *benchmark*, donde se inicia el servidor MySQL y se crea una tabla donde se harán los accesos. Tras esta fase de preparación, podemos lanzar las peticiones, que es lo que nos dará los datos relevantes en cuanto a prestaciones. Finalmente, podríamos eliminar la tabla de disco, esto no es estrictamente necesario ya que Gem5 no modifica la imagen de disco durante la simulación ya que al terminar se descartan por defecto [21]. Un ejemplo de *script* que haga todo lo descrito aquí se adjunta en el anexo B.

3.4.3. Trazas

Para la obtención de las trazas se utiliza NVMain. En los ficheros de configuración de este se pueden añadir *hooks* cuyo uso nos permite capturar las trazas. Para ello hay que definir dos parámetros en estos, «AddHook» y «PostTraceWriter», encargados de indicar al simulador si deseamos obtener una traza de las peticiones realizadas y la clase de NVMain a usar para ello. Los valores de estos campos son «PostTrace» y «NVMainTrace» respectivamente. Es recomendable añadir estas entradas precedidos de «;» con tal de que el simulador las ignore.

Como se ha comentado en el párrafo anterior, usamos la clase NVMainTrace para la obtención de las trazas. Esta clase viene por defecto con NVMain y genera trazas no muy grandes pero informativas y útiles para analizar. Las trazas que genera esta clase están formadas por varias líneas, cada una constituida por cinco campos:

- Ciclo en el que se ha realizado la petición.
- Qué tipo de petición es. Puede ser lectura «R» o escritura «W».
- Datos asociados en la petición. El tamaño de estos varía según el sistema que se esté simulando. Por ejemplo, si existen cachés el tamaño usual es de 64 bytes (ocho palabras de ocho bytes cada una).
- Datos almacenados previamente y que pueden ser modificados por la petición, en caso que sea una petición de escritura.
- Identificador de hilo que realizó la petición a memoria.

Para obtener el fichero (que será denominado «_ch0_rk0» y estará en el directorio «Config» de NVMain), simplemente lanzamos una simulación con un fichero de configuración de NVMain y dejamos que la ejecución se finalice, realizando un checkpoint de Gem5 justo en el momento previo de lanzar un *benchmark*. A partir de que concluya esta primera simulación, se lanza de nuevo pero esta vez restaurando desde el checkpoint con el fichero de configuración de NVMain modificado para que esta vez sí se obtenga la traza, es decir, con las líneas que hemos añadido descomentadas.

Gracias a los ficheros de trazas podemos obtener estadísticas que el simulador no ofrece de forma nativa. Por ejemplo, obtener el número de bloques de 64 *bytes* a los que se accede en cada página usada del *benchmark*, lo que permite realizar un análisis de la localidad de los accesos a nivel de página. Un *script* de ejemplo escrito en Python que realize esta tarea se puede encontrar en el anexo F. El análisis de los ficheros de traza también es útil para detectar si la aplicación de políticas de *prefetch* podría mejorar las prestaciones. En el capítulo 5.2 se entra en más detalle al respecto.

CAPÍTULO 4

Modelado de controladores e implementación de la propuesta

En los computadores actuales, los núcleos del procesador acceden a los datos de memoria principal mediante uno o varios controladores de memoria. Estos, son los encargados de dirigir las transacciones de datos mientras se aseguran del cumplimiento de protocolos, características específicas de memoria principal e incluso corrección y detección de errores.

El diseño e implementación de estos controladores determinan las latencias de acceso y la eficiencia del ancho de banda (cuánto ancho de banda se usa del total disponible). De la misma forma que dos procesadores soportan el mismo ISA pero implementados con diferentes microarquitecturas alcanzan diferentes prestaciones, dos controladores de memoria implementados de forma diferente pueden variar de manera significativa las prestaciones del sistema de memoria.

4.1 Modelado de controladores en el entorno utilizado

En este apartado se presenta la implementación de la clase base `MemoryController` del controlador de memoria de NVMain, que sirve como punto de inicio para poder explicar el controlador «FRFCFS» utilizado para la ejecución de los *benchmarks*.

4.1.1. La clase MemoryController

En el código de NVMain, los diferentes elementos deben heredar una clase base propia para ese elemento. En el caso de los controladores de memoria, esta clase se encuentra en el directorio «src» de NVMain y es el fichero «MemoryController.cpp», junto con su fichero de definiciones «MemoryController.h». En el código de estos ficheros se ofrecen un conjunto de funciones públicas con las que se interactúa con los controladores de memoria. También contienen funciones protegidas que solo pueden ejecutar la propia clase y sus clases derivadas. Estas sirven como apoyo a las funciones ya existentes y por ello no son públicas. Nos vamos a centrar en explicar las funciones públicas, que son las que

otorgan la funcionalidad clave, comentando las funciones auxiliares (las protegidas) según sea necesario para comprender mejor el código.

Las funciones públicas pueden repartirse en tres grupos. El grupo de funciones que sirven para inicializar o destruir el objeto, como son el constructor y las funciones «InitQueues» o «InitBankQueues». También podemos ver las funciones que implementan la funcionalidad del controlador que son «IsIssuable», «IssueCommand» y «RequestComplete». Finalmente, nos encontramos con el grupo de funciones que sirven para para obtener, entre otras, las estadísticas del controlador y configurarlo de la forma deseada. Algunos ejemplos de estas funciones son «SetConfig», «SetID» o algunas de las funciones «callback»[1] que se definen.

En la figura 4.1 podemos observar un diagrama de flujo que presenta el ciclo de vida de una petición de lectura o escritura en memoria que envía Gem5 a NVMain. De estos pasos, el controlador de memoria está relacionado en tres de ellos: «IsIssuable()», «IssueCommand()» y «RequestComplete()».

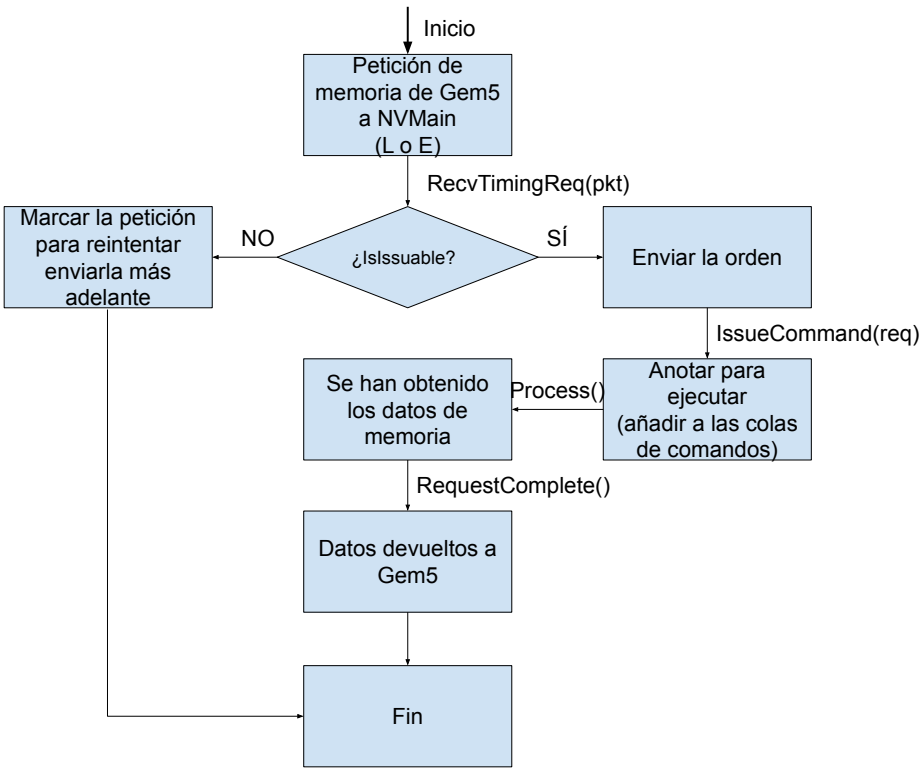

Figura 4.1: Ciclo de vida de una petición en NVMain

Las peticiones del simulador (Gem5) llegan al controlador de memoria (NVMain) mediante la clase nvmain_mem (en «Simulators/gem5» del directorio base de NVMain). Más concretamente, en la llamada a la función «recvTimingReq()».

[1] Una función callback es una función que se llama como respuesta a algún evento. En este caso, al evento «Refresh» y «Cleanup» del controlador de memoria.

La interacción que se realiza entre Gem5 y NVMain al llamar a «IsIssuable()» se muestra en la figura 4.2. Como se puede observar, en esta interacción Gem5 se cerciora de que se puede encolar la petición para su posterior proceso. Para ello, Gem5 realiza la llamada a IsIssuable (paso 1 en la figura), y NVMain responde (paso 2) en función del estado del controlador de memoria (p.e. sus recursos disponibles) y la información de la petición. En caso de que la petición no se pueda encolar, es marcada para un posterior reintento. Por otro lado, si la respuesta es positiva, Gem5 pasa seguidamente a realizar la interacción presentada en la Figura 4.3.

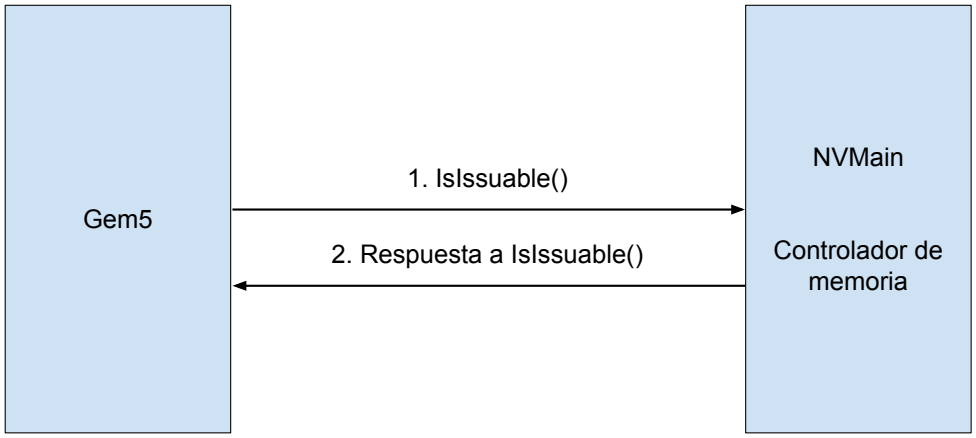

Figura 4.2: Interacción entre controlador de memoria y simulador al realizar la llamada «IsIssuable()»

En la figura 4.3, se observa como Gem5 indica a NVMain, a través de la llamada a «IssueCommand()» (paso 1), que acepte la petición en el controlador. Como consecuencia de esta llamada, NVMain genera eventos que se dispararán en futuros ciclos de la simulación (dirigida por eventos) de Gem5, cuando la petición sea completada. Estos eventos se registran en la cola de eventos de Gem5 (paso 2). Finalmente, NVMain devuelve el control a Gem5 (paso 3) para que este último pueda continuar con la simulación del sistema (es decir, con la gestión de los eventos generados por todos los componentes del sistema), lo que eventualmente, llevará a la interacción presentada en la figura 4.4 cuando el evento generado por NVMain se dispare.

En la interacción de la figura 4.4, Gem5, al dispararse el evento correspondiente, llama a la función «RequestComplete()» en NVMain (paso 1). Como respuesta a esta llamada, NVMain devuelve el resultado de la petición completada (paso 2).

4.1.2. El controlador de memoria FRFCFS

Una vez conocemos las bases tras las que operan los diferentes tipos de controladores de memoria en NVMain, podemos adentrarnos a discutir conceptos más específicos. En este caso comentamos el funcionamiento del controlador «FRFCFS».

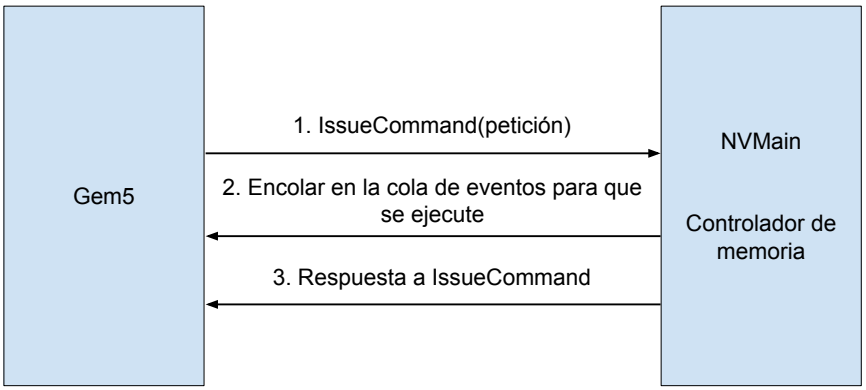

Figura 4.3: Interacción entre controlador de memoria y simulador al realizar la llamada «IssueCommand()»

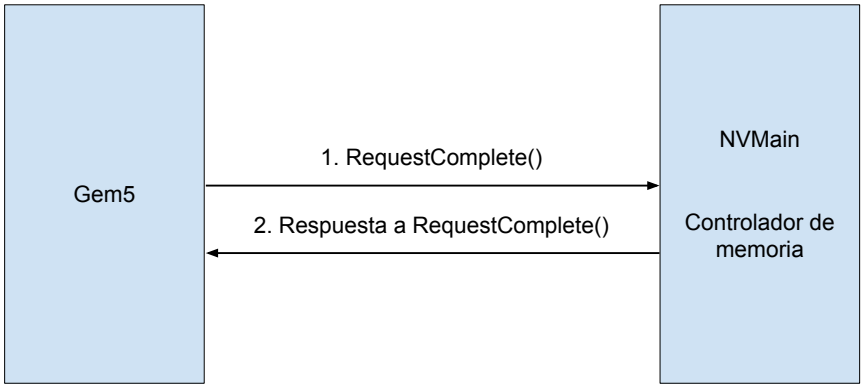

Figura 4.4: Interacción entre controlador de memoria y simulador al realizar la llamada «RequestComplete()»

En el controlador «FRFCFS», el método «IsIssuable()» sigue la siguiente lógica. Si la cola de memoria tiene encoladas 32 peticiones retorna el valor «*false*». En cualquier otro caso sí se permite añadir peticiones adicionales a la cola, devolviendo el valor «*true*».

Respecto al método «IssueCommand()», simplemente comprueba -de nuevo- que se puede encolar llamando a su propio método «IsIssuable()». Tras ello, añade la petición «*request*» al final de la cola de peticiones. Esto se debe a que, al implementar FRFCFS[2] todas las peticiones se deben añadir al final de la cola. Tras encolar la petición, comprueba si existen peticiones planificables para el próximo evento «*wakeup*» del simulador; si no existen, espera a que llegue el momento. Antes de devolver el valor de retorno, ac-

[2]El algoritmo FRFCFS (*First-ready, first come, first served*) va lanzando las peticiones a los módulos priorizando los aciertos (first-ready) en el row buffer o buffer de filas). Si no existen «peticiones listas», simplemente sigue la política FCFS, es decir ejecuta la primera que entró a la cola, las nuevas peticiones se añaden al final de la cola.

tualiza la cantidad de peticiones de tipo *READ* o *WRITE* según el tipo de la petición, incrementando el contador que mantiene las correspondientes estadísticas.

Para concluir esta explicación sobre el controlador de memoria FRFCFS, vamos a explicar el método «RequestComplete()», una reimplementación del mismo método de la clase base MemoryController. En esta nueva implementación del método, «FRFCFS» comprueba si las peticiones de tipo *WRITE* y *WRITE_PRECHARGE* han sido canceladas o pausadas y las vuelve a encolar sí así ha sido, pero esta vez al principio de la cola dando por lo tanto prioridad a este tipo de peticiones que han sido interrumpidas o canceladas. Después simplemente comprueba si la petición es de tipo *READ*, *READ_PRECHARGE*, *WRITE* o *WRITE_PRECHARGE*. Si ese es el caso, actualiza algunas estadísticas y llama al método «RequestComplete()» de MemoryController, cuyo valor de retorno será el valor de retorno del método «RequestComplete()» de la instancia del controlador FRFCFS.

4.2 Propuesta

El objetivo de este proyecto es implementar y probar un controlador sencillo para memorias del tipo NVMM que mejore las prestaciones. Para realizarlo en los simuladores descritos, se requiere un aprendizaje previo de estos simuladores. En las próximas secciones se describen los cambios que se realizan sobre el controlador de memoria FRFCFS. También se explican los detalles de implementación, así como los pasos a seguir para añadir el controlador propuesto a los simuladores y poder evaluar sus prestaciones.

En particular, se añade una memoria caché en el controlador de memoria para aumentar las prestaciones del controlador, y por consiguiente, del subsistema de memoria. Debido a estar en un nivel bajo de la jerarquía, las cachés deben ser de gran tamaño. Para estudiar los posibles beneficios que se alcanzarían por la caché gracias a la explotación de la localidad de los datos, los experimentos utilizan una cache de 256 MB con tamaño de bloque de 64 B y reemplazo aleatorio.

4.2.1. Creación de la clase FRFCFS_CACHE

Para añadir la caché, necesitamos nuestro propio controlador de memoria. Para esto, tenemos que realizar un conjunto de pasos para que NVMain reconozca el controlador, lo compile y pueda crear objetos de esta clase a la hora de la compilación de Gem5 y NVMain. El primer paso para esto es obtener una copia del controlador de memoria FRFCFS, para ello ejecutamos las siguientes órdenes (desde el directorio base de NVMain):

```
cd MemControl/
cp FRFCFS/ FRFCFS_CACHE/
```

Después de realizar la copia, procedemos a renombrar la clase y el fichero de definiciones usando los siguientes comandos:

```
cd FRFCFS_CACHE
```

```
mv FRFCFS.h FRFCFS_CACHE.h
mv FRFCFS.cpp FRFCFS_CACHE.cpp
```

Una vez hemos realizado el cambio de nombres, modificamos primero el fichero «FRFCFS_CACHE.h». Para ello debemos realizar las siguientes modificaciones.

1. Cambiar la línea 34 y 35 por «#ifndef __FRFCFS_CACHE_H» y «#define __FRFCFS-_CACHE_H» respectivamente.
2. Actualizar el nombre de la clase en la línea 42 a «FRFCFS_CACHE».
3. Actualizar el constructor (línea 45) a «FRFCFS_CACHE( );» y el destructor (línea 46) a « FRFCFS_CACHE( );».

Tras esto, debemos actualizar el fichero de extensión «.cpp». En este realizaremos los cambios que se indican a continuación:

1. Incluimos el fichero de definiciones que hemos modificado. Sustituimos la línea 34 por «#include "MemControl/FRFCFS_CACHE/FRFCFS_CACHE.h"»
2. Cambiamos el constructor y el destructor en la línea 52 y 85 respectivamente, sustituyendo el «FRFCFS» a la derecha de los dos dobles puntos por «FRFCFS_CACHE».
3. Modificaremos las lineas 87 y 105 para que en vez de «FRFCFS» sea «FRFCFS_-CACHE».
4. Tenemos que actualizar las cabeceras de todos los métodos. Cada método del fichero indica a qué clase pertenece (mediante el identificador que está a la izquierda de los dos dobles puntos). Todos estos identificadores que indican «FRFCFS» hay que reemplazarlos por «FRFCFS_CACHE».

A continuación, hay que modificar el fichero «SConscript», este fichero es el encargado de avisar a NVMain donde están nuestros ficheros fuentes. Para esto simplemente tenemos que sustituir el parámetro de la función NVMainSource en la línea 42 por «FRFCFS_CACHE.cpp». Estos son todos los cambios necesarios para tener las clases creadas y al compilador avisado de que queremos usarlas.

Ahora debemos modificar el fichero «MemoryControllerFactory.cpp», ubicado en el directorio «MemControl», al que se puede acceder desde el directorio base de NVMain. En este fichero hay que realizar los siguientes cambios:

1. Debemos añadir el fichero de definiciones, para ello hay que añadir la siguiente directiva «include» en la línea 43: «#include "MemControl/FRFCFS_CACHE-/FRFCFS_CACHE.h"».[3]

[3]Remarcar que las comillas usadas en la directiva include son comillas dobles inglesas y no las españolas, que son las que rodean la línea a insertar.

2. Tras la línea 61 añadiremos «else if (controller == "FRFCFS_CACHE" | | controller == "FRFCFS-CACHE") memoryController = new FRFCFS_CACHE( );». La instrucción «new FRFCFS_CACHE( );» puede ubicarse en la siguiente línea, siguiendo el estilo del bloque «else if».

Una vez hemos hecho este cambio, que se encarga de crear nuestra clase cuando se encuentra el valor «FRFCFS-CACHE» o «FRFCFS_CACHE» en el fichero de configuración de NVMain, debemos ejecutar el *script* «GenerateSConscript.sh» ubicado en el directorio «MemControl». Tras ejecutarlo, NVMain será capaz de compilar y reconocer nuestro controlador de memoria. Todos estos pasos que hemos realizado son necesarios solo la primera vez que se añaden los ficheros y no es necesario repetirlos cuando se realizan cambios al código del controlador.

4.2.2. Implementación de caché en el controlador

Una vez tenemos los anteriores pasos hechos, podemos proceder a implementar la memoria caché. Esta es muy sencilla y se basa en un diccionario (en inglés, a esta estructura de datos también se le conoce como *map*; por lo tanto, usaremos ambos términos indistintamente) y unas funciones para leer y escribir datos en este. El código de esta implementación se puede consultar en el anexo C para el fichero de definiciones y en el D para la implementación en C++.

El diccionario está formado por claves de tipo «uint64_t» y cuyo valor es un vector de clase MySRAMCacheEntry, que es una nueva clase definida en el fichero C. Esta clase representa un byte de la entrada en caché, y contiene el propio byte de información. Esto en principio no parece útil ya que como hemos comentado en el capítulo 2.1.2, las peticiones por norma general son de 64 bytes pero el simulador Gem5 puede realizar alguna con menor tamaño, como podrían ser ocho bytes para leer una única palabra. Esto ocurre en la simulación FS, donde al realizar el arranque del sistema se acceden a direcciones específicas con tamaños de lectura diferente a 64 B. Además, las peticiones de 64 B que genera Gem5 no tienen que estar alineadas. Puede darse el caso de que una petición, por ejemplo, acceda a la dirección «0x1000010»,[4] esta dirección no está alineada a 64 bytes ya que (0x1000010 mod 0x40) tiene como resultado un valor diferente a 0. Deducimos por lo tanto que esta petición quiere obtener 48 bytes de la línea de caché a la que accede y 16 bytes de la siguiente.

La clase MySRAMCacheEntry también ofrece métodos para manipular los objetos de esta clase. Tanto para leer («getDato()») o escribir («setDato()») el byte como para modificar el valor del bit de válido («setVal()» y «resetVal()») o para leerlo («getVal()»).

[4] El «0x» al comienzo del número denota que este es un número hexadecimal.

La clase MySRAMCache

Como hemos comentado al principio de este capítulo, la clase MySRAMCache contiene toda la lógica necesaria para implementar la caché asociativa con reemplazo random. Esta utiliza un diccionario como base para implementar el funcionamiento.

El funcionamiento de la caché MySRAMCache es sencillo. A partir de una dirección emitida por el procesador, la caché obtiene la etiqueta y el desplazamiento dentro de ese bloque de 64 B. La etiqueta se usa como clave del *map* con los datos, mientras que el desplazamiento sirve para conocer el bloque en el que comenzará a leer o escribir. Estos valores se calculan mediante las máscaras «mascaraDir» y «mascaraDesplz», gracias al operador «&» de C++ (este realiza la operación lógica «AND» a nivel de bit).

La clase MySRAMCache ofrece las cuatro operaciones básicas para un objeto. Un constructor para crear una instancia de la clase, una función para leer datos de la caché, otra para escribir/actualizar datos de la caché y una última función para eliminar una línea de caché. Estas funcione son, respectivamente, «MySRAMCache::MySRAMCache()», «MySRAMCache::readData()», «MySRAMCache::writeData()» y «MySRAMCache::invalidateData()». También hemos añadido el método de utilidad «MySRAMCache::hasData()», que sirve para comprobar si existen datos válidos en las líneas a las que se va a acceder y por lo tanto se puede usar para comprobar si hay un acierto, así como *getters* y *setters* para las variables «maxSize» y «currSize» y un método *getter* adicional para obtener la latencia de la caché. Esta interfaz se puede consultar en el anexo C.

A continuación, y tras haber comentado un poco la interfaz principal que ofrece nuestra clase «MySRAMCache», vamos a explicar algunos detalles de implementación de las funciones principales de «MySRAMCache.h». Para hacer frente al problema de los accesos con direcciones no alineadas, utilizamos un bucle *while* en el cual se realiza el acceso de escritura o lectura dependiendo de la función a una única línea. En este acceso se tiene en cuenta el posible desplazamiento, así como cuántos bytes faltan por acceder del total. Si durante este acceso se llega al final de la línea y todavía quedan datos por leer, haremos otra iteración del bucle, esta vez con la etiqueta y el desplazamiento actualizados a la siguiente línea.

Un ejemplo de esto puede observarse en la función «MySRAMCache::readData()» en el anexo D. El bucle *while* inicial se asegura de que antes de acabar la lectura hemos accedido a tantos bytes como nos pida la petición. A continuación, a partir de la etiqueta y desplazamiento que obtenemos usando las máscaras, accedemos al diccionario con la etiqueta que nos devuelve la línea a leer. El acceso a la línea empieza en el byte indicado por el desplazamiento, y se irá leyendo byte a byte y añadiéndolos al vector de retorno «rv» hasta que lleguemos al final de la línea o no tengamos que leer más datos. Tras esto, actualizamos la dirección de acceso «moddedAddr» con la cantidad de datos que hemos leído más uno (para indicar que queremos empezar a leer la próxima dirección), ya que en caso de que falten bytes por leer queremos acceder a la línea siguiente, con etiqueta desplazada hasta el inicio de la siguiente línea, y así asegurarnos de que accederemos a la línea adecuada. Seguidamente, repetimos toda la operación pero esta vez el bucle *while* acabará, ya que habremos leído los bytes necesarios para la petición.

Desde el punto de vista de implementación tanto «hasData()» como «readData()» son muy similares. Ambas funciones acceden al *map* con tal de leer los datos, pero las finalidades de ambas son algo diferentes. La función «hasData()» se asegura de que los datos accedidos sean válidos en la dirección que le proporcionamos. Mientras que «readData()» tiene como finalidad retornar todos los datos que existen en la línea (o líneas) a las que se acceden en la petición de memoria. Esta distinción entre ambas funciones es crítica ya que a la hora de implementar otros algoritmos de reemplazo, como los mencionados en 2.1.2, las acciones que se deban tomar en el momento de la lectura de cache deberán de realizarse en «readData()» y no en «hasData()», que debería usarse solo para distinguir entre un acierto o fallo en caché.

Por otra parte, «writeData()» tiene un funcionamiento algo diferente. Esto se debe a que a la hora de escribir datos debe controlar si caben o no en caché y, dependiendo de esto, reemplazar líneas si es necesario. Es por ello que «writeData()» está formado por dos casos diferenciados. El caso en el que el bloque ya se encuentre almacenado y el caso en el que no se encuentre. En el primero, simplemente obtiene el bloque almacenado y lo actualiza con los valores correspondientes. En el segundo caso, crea una entrada nueva y escribe en esta los valores que se quieren guardar en caché y a continuación procede a comprobar si hay que realizar un reemplazo dependiendo de si la caché está llena o no. Si no hay que realizar un reemplazo simplemente insertamos la línea. Sin embargo, en caso de tener que realizar un reemplazo antes de poder insertar la nueva línea, hay que identificar la línea víctima a ser reemplazada y eliminar esa línea mediante el método «invalidateData()». Finalmente, se añade la nueva línea.

Todas estas operaciones que hemos mencionado las realizamos mientras mantenemos todas las variables de la caché actualizadas y con valores coherentes. La finalidad de esta tarea es encontrar un estado consistente y un código fácil de entender, consiguiendo por lo tanto una depuración de la caché lo más sencilla posible. En caso de ser necesaria una depuración del código recomendamos la compilación y uso del binario «gem5.debug». El motivo de esta elección se debe a que los diferentes tipos de binario que Gem5 ofrece (estos se mencionan al final del capítulo 3.1.1) poseen un grado de optimización diferente, siendo el aquí mencionado el menos optimizado y por lo tanto el más fácil de usar en el momento de la depuración. Una vez realizada la compilación del binario «gem5.debug» se pueden usar aplicaciones de depuración como puede ser GDBGUI que añade una interfaz gráfica al depurador GDB y es la que hemos usado en este proyecto.

4.2.3. Integración de la caché en el controlador FRFCFS_CACHE

Una vez tenemos la implementación de la caché, falta que la integremos en el controlador. Para ello primero tenemos que modificar el fichero de definiciones del controlador FRFCFS_CACHE. Tras esto, vamos a introducir cambios el código fuente de la su clase. Finalmente modificaremos el fichero SConscript de tal forma que «SCons» compile el código de la caché a la hora de crear el binario Gem5-NVMain.

Para poder crear la caché en el controlador debemos usar una directiva «include» para importar el fichero de definiciones y poder crear un objeto. Por ello recomendamos mo-

ver tanto «MySRAMCache.h» como «MySRAMCache.cpp» a un directorio en la carpeta que contiene el controlador, en nuestro caso esta carpeta la conocemos como «MySRAMCache». Una vez hemos movido los ficheros a este nuevo directorio podemos modificar «FRFCFS_CACHE.h» entre las líneas 37 y 38 con la siguiente directiva «include» para incluir la caché:

```
#include "MemControl/FRFCFS_CACHE/MySRAMCache/MySRAMCache.h"
```

Seguidamente, podemos crear una variable de tipo «MySRAMCache» con nombre «DataCache» y que sea un puntero. A continuación creamos una instancia de la clase usando el constructor y cuyos parámetros serán «22» y «4». También necesitamos tres variables como estadísticas de la caché. Estos cambios se podrían realizar entre la línea 73 y 74. Un ejemplo podría ser:

```
/* Cache */
MySRAMCache *DataCache = new MySRAMCache(22, 4);
/* Cache Stats */
uint64_t myCacheTries, myCacheHits, myCacheWrites;
```

Tras ello, modificaremos el código fuente del controlador de memoria para aprovechar esta caché. Realizaremos cuatro modificaciones, una en el constructor de la clase FRFCFS_CACHE para instanciar los contadores de estadísticas. La segunda en el método «FRFCFS_CACHE::RegisterStats()» con el objetivo de añadir las variables como estadísticas de las que NVMain se encargará de mostrar al final de la ejecución. La tercera modificación la haremos en «FRFCFS_CACHE::RequestComplete()» que se encargará de escribir los datos en la caché. Finalmente cambiaremos «FRFCFS_CACHE::IssueCommand()» que explotará los datos en caché para mejorar las prestaciones así como realizar la escritura en caché de las peticiones «WRITE».

Las modificaciones para contabilizar las estadísticas son muy sencillas. Empezaremos por inicializar las variables de estadísticas. Entre las líneas 78 y 79 del fichero «FRFCFS_CACHE.cpp» (es decir, en el constructor) las inicializaremos a cero tal que:

```
myCacheTries = 0;
myCacheHits = 0;
myCacheWrites = 0;
```

A continuación modificaremos la función «FRFCFS_CACHE::RegisterStats()», esta modificación se puede llevar a cabo entre las líneas 124 y 125 con el siguiente código:

```
AddStat(myCacheTries);
AddStat(myCacheHits);
AddStat(myCacheWrites);
```

Tras estas modificaciones sencillas, podemos comenzar a implementar la caché en el código del controlador. La modificación que realizamos en el método «RequestComplete()» la insertamos entre las líneas 202 y 203. Esta comprobará que los datos que contiene la petición son válidos y en caso de que lo sean, los añadirá a caché solo si la petición no es de tipo «WRITE». Esto evita posibles problemas de consistencia en caché al escribir dos veces la misma petición. El código que realiza esto es:

```
if(request->type != WRITE)
{
    if(request->data.IsValid())
    {
        DataCache->writeData(request->address.GetPhysicalAddress(),
                             request->data.rawData,
                             request->data.GetSize());
    }
}
```

Seguidamente modificaremos el método «IssueCommand()». En este, las lecturas y escrituras se tratan de forma diferente. Las peticiones de lectura intentarán obtener los datos de la caché y si los encuentra (hasData() devuelve el valor «*true*»), se añadirán a la cola de eventos para ejecutarse tras 4 ciclos, ya que es la cantidad de retardo que hemos definido en el constructor de la caché. Por otro lado, todas las peticiones de escritura modifican tanto la caché como memoria principal. Esta política de acceso en escritura se conoce como «write-through». Antes de llevar a cabo la modificación, realizamos la misma comprobación sobre la validez de los datos que en «RequestComplete()». Este cambio en el método «IssueCommand()» se realizará entre las líneas 159 y 160, siendo el código necesario para implementarlo:

```
if( req->type == READ )
{
    ++myCacheTries;

    uint64_t issueReqAddress = req->address.GetPhysicalAddress();
    uint64_t issueReqSize = req->data.GetSize();
    if(DataCache->hasData(issueReqAddress,
       issueReqSize))
    {
        ++mem_reads;
        ++myCacheHits;
        uint64_t proximoAcceso = req->arrivalCycle +
                                 DataCache->getLatenciaCiclos();

        req->data.rawData = DataCache->readData(
                             issueReqAddress,
```

```
                            issueReqSize);

            assert(req->data.rawData != nullptr);

            GetEventQueue()->InsertEvent(EventResponse, this, req,
                                         proximoAcceso);
            return true;
        }
    }
    else
    {
        if(req->data.IsValid())
        {
            ++myCacheWrites;
            DataCache->writeData(req->address.GetPhysicalAddress(),
                                 req->data.rawData,
                                 req->data.GetSize());
        }
    }
```

Antes de poder recompilar el simulador con las modificaciones hechas, debemos indicar a SCons que debe compilar también los fuentes de la clase que representa nuestra caché. Esto se consigue añadiendo, en el fichero «SConscript» ubicado en el directorio base de NVMain la siguiente llamada a función en la línea 62:

```
NVMainSource('MemControl/FRFCFS_CACHE/MySRAMCache/MySRAMCache.cpp')
```

Finalmente, procedemos a compilar de nuevo el simulador Gem5-NVMain con la siguiente orden (desde el directorio base de Gem5):

```
scons build/X86/gem5.opt EXTRAS=../NVmain -j $(nproc)
```

Desde este momento el controlador «FRFCFS_CACHE» está completamente implementado y se puede usar modificando el valor del campo «MEM_CTL» de los ficheros de configuración de NVMain por «FRFCFS_CACHE» o «FRFCFS-CACHE».

Recordar que existe el repositorio en github[5] que contiene los cambios discutidos.

[5]https://github.com/miavgu/gem5-nvmain-tfg Último acceso 24 de agosto de 2021.

CAPÍTULO 5
Evaluación de resultados

En este capítulo se discuten los resultados obtenidos de las cargas de trabajo. Comenzaremos por discutir los resultados de los *benchmarks* analizando los efectos que han tenido las modificaciones sobre las prestaciones. Tras ello discutiremos el uso de las trazas, cuyo uso mediante *scripts* de interpretación nos ofrecerá información adicional más allá de la que el simulador puede ofrecer por defecto. Todos los resultados que vamos a analizar y discutir los hemos obtenido mediante la ejecución de simulaciones FS con el binario Gem5-NVMain.

5.1 Análisis de prestaciones

En este apartado se comparan y analizan los resultados del controlador FRFCFS y del controlador FRFCFS_CACHE. Para ello se han utilizado los diferentes *benchmarks* estudiados en el apartado 3.4.1. Para la obtención de las estadísticas se han realizado las mismas simulaciones únicamente modificando el controlador de memoria. Por ello se han utilizado los *scripts* descritos en los anexos A y B con la configuración del simulador descrita al final del capítulo 3.3 y una caché con 2^{22} líneas de 64 B de tamaño, el exponente del número de líneas es el primer parámetro del constructor de la caché. Esto ofrece un tamaño total de 256 MiB.

En el caso de Redis (tabla 5.1), los resultados que se muestran son el número de peticiones por segundo que el sistema es capaz de realizar, como se puede observar en la tabla existen cuatro tests, correspondientes con operaciones básicas de Redis que son «set», «get», «lpush» y «lpop». Para cada uno de estos tests se ha ido aumentando la cantidad de peticiones totales que debe hacerse de cada tipo con tal de cargar el sistema lo máximo posible. A partir de los datos expuestos en esta tabla se puede calcular el *speedup* obtenido debido al uso de la caché, que varía entre un 0.6 % y un 3.5 %. Esto nos indica que el incorporar la caché al controlador ha aumentado las prestaciones del sistema para esta aplicación.

Del *benchmark* MySQL (tabla 5.2) se muestran las transacciones por segundo realizadas a lo largo de la duración de la prueba. Para la obtención de estos datos fijamos la cantidad de tiempo simulado que queremos que dure el *benchmark*, en nuestro caso

Controlador de memoria	Test	Número total de peticiones			
		10000	**25000**	**50000**	**100000**
FRFCFS	SET	6983.0	7050.0	6957.0	7098.0
	GET	7047.0	7108.0	7017.0	7122.0
	LPUSH	6983.0	6987.0	6994.0	7119.0
	LPOP	6983.0	6985.0	6955.0	7052.0
FRFCFS_CACHE	SET	7074.0	7130.0	7202.0	7140.0
	GET	7163.0	7215.0	7221.0	7173.0
	LPUSH	7204.0	7194.0	7234.0	7191.0
	LPOP	7087.0	7130.0	7189.0	7136.0

Tabla 5.1: Peticiones por segundo de los cuatro tests del *benchmark* Redis para los controladores de memoria estudiados, variando el número total de peticiones realizadas a memoria por test.

hemos realizado una prueba con 30 segundos y otra con 60 segundos, y Sysbench se encargará de detener la prueba y devolvernos los resultados al fin del periodo de tiempo establecido. Si calculamos el *speedup* al igual que con Redis podemos observar una mejoría en las prestaciones de la aplicación, pero a diferencia de este último, esta ganancia es más acentuada, entre un 9.7 % y un 10.2 %.

Para comprender mejor esta notoria diferencia en el *speedup* conseguido en ambas pruebas, es importante fijarnos en el hit ratio de la caché del controlador. El hit ratio indica la tasa de acierto (bloques que se encuentran en caché cuando la CPU ha realizado una petición de memoria) que se han realizado a lo largo de las pruebas. Esta medida se puede usar como una muestra de lo eficaz que ha sido la caché. Se puede calcular de forma sencilla conociendo la cantidad de intentos o *tries* que se han hecho a la caché frente a los aciertos o *hits*. En el capítulo 4.2.3, añadimos estas estadísticas con tal de que NVMain las devolviese actualizadas al final de las diferentes simulaciones realizadas. Gracias a estos datos, podemos obtener los hit ratios de las diferentes simulaciones, que se encuentran recopilados en la tabla 5.3.

Como podemos observar en la tabla 5.3, el hit ratio de las ejecuciones del *benchmark* MySQL es aproximadamente el doble que las ejecuciones de Redis. Esto revela que MySQL ha hecho un uso de caché mucho mayor en comparación a Redis y por lo tanto estas ejecuciones se han visto más beneficiadas de accesos a memoria con menos latencia, dando así una aceleración superior. También podemos observar como, en el mejor de los casos, el porcentaje de utilización de la caché no supera un 46 %; en el capítulo 6.3 se comentan algunas formas de poder aumentar esta utilización que se dejan como posible trabajo futuro a realizar.

5.2 Análisis de uso de bloques

Cada vez que se incorpora un nuevo nivel en la jerarquía, una de las principales decisiones de diseño es decidir el tamaño de bloque. Por ejemplo, el tamaño de bloque que

Controlador de memoria	30 s	60 s
FRFCFS	1160.7	1127.3
FRFCFS_CACHE	1273.5	1242.0

Tabla 5.2: Transacciones por segundo del *benchmark* MySQL para cada controlador de memoria, variando el número de segundos simulados.

Redis				MySQL	
10000	**25000**	**50000**	**100000**	**30 s**	**60 s**
23.28 %	22.65 %	24.12 %	23.07 %	45.83 %	45.96 %

Tabla 5.3: Tasa de acierto en la caché del controlador para los diferentes *benchmarks* ejecutados.

se busca en memoria cuando hay un fallo de cache ha ido cambiando a lo largo de la evolución de los computadores. Inicialmente, algunas cache tenían un tamaño de bloque de unas pocas palabras (por ej. 16 B), posteriormente al aumentar la discrepancia entre la velocidad de la memoria principal y la caché del procesador, el tamaño aumentó a 32 B y actualmente el tamaño se ha estandarizado en 64 B. Asimismo, algunos procesadores con alta presión en la jerarquía, como el IBM Power4 incorporan un tamaño de bloque de la cache de nivel 3 de 256 B manteniendo el tamaño de bloque del resto de los niveles de caché en 64 B. Es lo que se conoce como caché de sectores, que es una caché donde el tamaño de bloque (o de sector, en el contexto de estas cachés) es un múltiplo entero (por ej. cuatro) del tamaño de bloque de las cachés de los niveles inferiores. A nivel conceptual, los cuatro bloques almacenados en un sector comparten la etiqueta pero cada uno dispone de un bit de válido específico para indicar si se encuentra el bloque y un bit de modificado para indicar si el contenido es distinto del almacenado en el nivel de caché inmediatamente superior.

Puesto que debido a la presión de memoria suele ser necesario aumentar el tamaño de bloque en niveles inferiores de la jerarquía de memoria, es muy probable que sea necesario utilizar una caché de sectores en nuestro controlador de memoria. Para explorar el tamaño de sector óptimo, es necesario realizar un estudio estadístico y analizar el número de bloques accedidos en cada sector. Por ejemplo, si el procesador solo pide un bloque de 64 B dentro de cada sector de 256 B, sería desaconsejable establecer 256 B como unidad de acceso al módulo NVRAM, ya que esta acción desperdiciaría espacio en la caché y no se traduciría en un aumento de prestaciones. Por contra, si se accede un 75 % de los bloques almacenados en la caché, esto se traducirá en una reducción del número de accesos al módulo NVRAM, se amortizará el tiempo de acceso y, en consecuencia, se obtendrá una ganancia de prestaciones.

Modificar el simulador para poder acceder a estos datos de uso de los sectores es una tarea costosa y por ello se usan *scripts* para obtener este tipo de información. Gracias a estos *scripts* que se ejecutan sobre las trazas obtenidas de las ejecuciones, es posible realizar el estudio teórico analizando los bloques de manera agrupada según el tamaño del sector. Es por ello que se presenta este análisis para diferentes tamaños de sector,

desde 256 B hasta 4096 B, que coincide con el tamaño de página, divididos en bloques de 64 B.

La localidad de los datos puede variar mucho según la aplicación. Por ejemplo, las aplicaciones multimedia y las basadas en matrices tienen una mayor utilización de los datos dentro de los bloques de la página, por lo que presentan una alta localidad espacial. Por contra, aplicaciones irregulares en el acceso a los datos, como la mayoría de las aplicaciones SPEC de enteros presenta un localidad mucho más pobre. Por tanto, para el diseño de la cache y, con miras a elegir el tamaño de bloque óptimo se precisa analizar las características de las aplicaciones destino que se utilizarán en la plataforma. En este proyecto nos basamos en una traza obtenida del *benchmark* Redis con 10000 peticiones en cada uno de los tests.

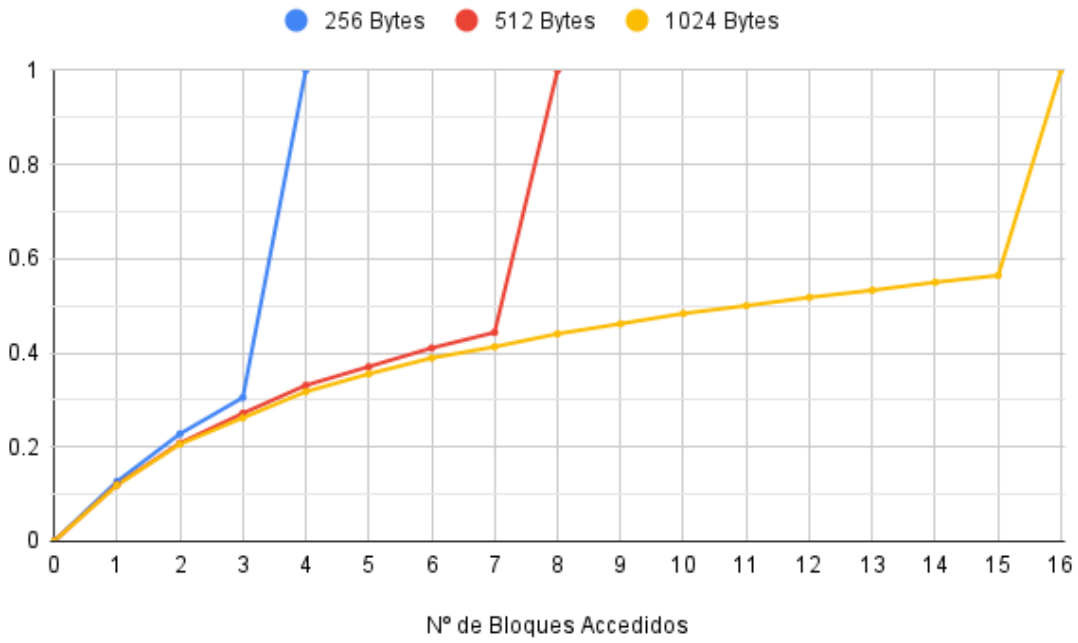

Figura 5.1: Función de distribución (CDF) de bloques accedidos por sector, para sectores de 256 B, 512 B y 1 KB.

Un tamaño de sector mayor presenta sus ventajas e inconvenientes. En principio, un mayor tamaño permite explotar mejor la localidad espacial de los bloques, reduciendo de esta manera el número de accesos al módulo NVM y, en consecuencia, la penalización de acceso. Sin embargo, un tamaño excesivo puede conllevar una baja utilización de los bloques. Es decir, es posible que muchos de sus datos no sean accedidos debido a que aumenta la fragmentación. Esto depende en gran medida del tipo de aplicaciones, por ello, debemos realizar este estudio sobre las aplicaciones analizadas.

Los resultados del estudio se aprecian en las figuras 5.1 y 5.2 que muestran la función de distribución variando el tamaño del sector. Los números en el eje X representan el número de bloques de 64 accedidos. Este *benchmark* muestra una elevada localidad espacial, donde todos los bloques son accedidos. Por ejemplo, para un tamaño de sector 256 B, por

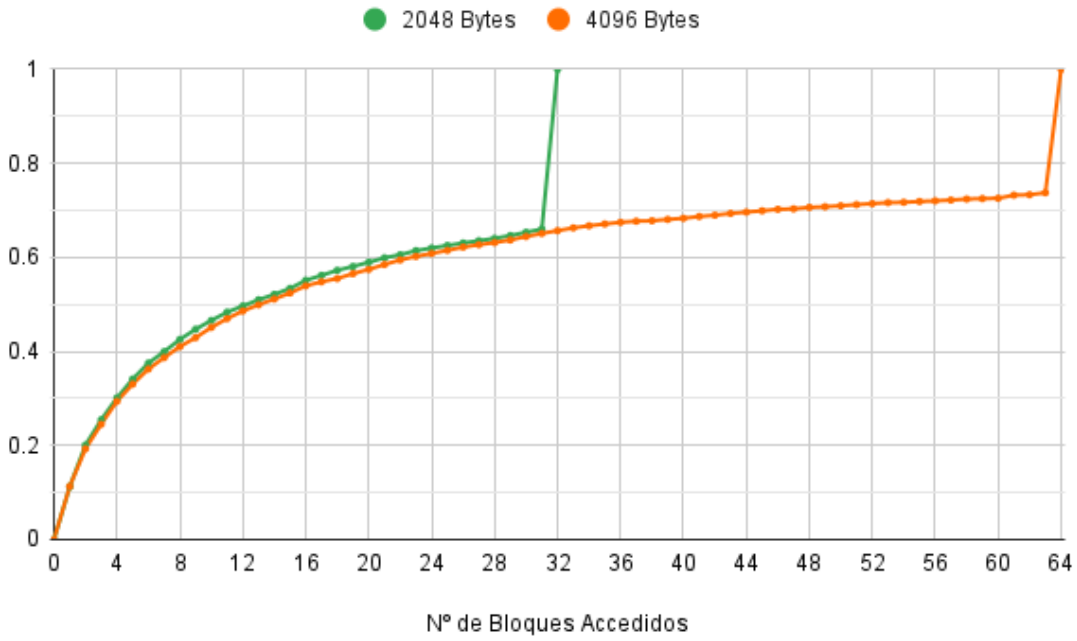

Figura 5.2: Función de distribución (CDF) de bloques accedidos por sector, para sectores de 2KB y 4KB.

término medio, en un 12 % (0.12 en el eje Y de la figura) de los sectores se accede a un solo bloque (es decir, menos del 25 % del sector). Sin embargo, para un sector de tamaño 1024 B, podemos observar como el porcentaje de sectores donde se accede a menos de un 25 % (4 bloques o menos) de sus bloques sube hasta el 32 %.

Si el análisis se realiza en términos absolutos se aprecia que en sectores de 256 B se accede a sus 4 bloques un 70 % de las veces, mientras para sectores 512 B se accede a los 8 bloques del mismo un 53 % de las veces, mostrando todavía una alta localidad espacial. Más aún, esta localidad se mantiene incluso para tamaños de sector de 4 KB, donde por término medio, se accede a todos los 64 B del sector alrededor de un 28 % de las veces.

Esta alta localidad muestra que los prefetchers podrían funcionar de manera excelente con tamaño de sector tan grandes. Estos prefetchers son necesarios para amortizar la alta latencia de acceso a la NRAM. Nótese que un tamaño de bloque de 256 B podría aportar beneficios escasos, máxime teniendo en cuenta que la latencia de acceso a la NVRAM es alrededor de 10 veces mayor que la DRAM. Sin embargo, un tamaño de sector más grande tiene como inconveniente un mayor tiempo de ocupación del bus accediendo al módulo NVRAM, lo que provocaría una mayor contención en el acceso.

Para estudiar el compromiso entre tamaños de sectores más grandes compuestos con un mayor número de bloques que reduciría la tasa de fallos y su mayor contención que afectaría adversamente a las prestaciones es necesario una simulación detallada del sistema. Para ello, se necesita modelar la lógica de integración del controlador con su caché, los módulos DRAM y los módulos NVRAM. Estas modificaciones se plantean como trabajo futuro.

CAPÍTULO 6

Conclusiones

A continuación, y como conclusión de este TFG, comentamos las principales ideas que extraemos de este trabajo. Para ello, comprobamos si hemos cumplido los objetivos propuestos en 1.2. Tras ello, expondremos la relación de este TFG con los estudios cursados, tanto de la rama de ingeniería de computadores como con el grado en general. Finalmente concluiremos con posibles ampliaciones a realizar sobre este trabajo de cara a futuros trabajos de final de máster universitario.

6.1 Objetivos alcanzados

En esta sección se expondrán de nuevo los objetivos que planteamos en 1.2, en base a estos, expondremos si se han llevado a cabo o no de forma satisfactoria junto con una explicación.

- **Familiarizarse con la simulación de sistemas con memoria NVMM, entendiendo el código que modela su funcionamiento y modificándolo de acuerdo con los restantes objetivos del proyecto.** Este objetivo se ha llevado a cabo de forma satisfactoria ya que hemos podido añadir un componente de creación propia al simulador y que funcione el simulador con el nuevo componente de forma correcta.

- **Elaborar y diseñar un controlador de memoria principal para NVRAM DIMM que incluya una memoria cache para mejorar las prestaciones.** Gracias a la inclusión de la caché que hemos implementado en el controlador «FRFCFS» ya existente y que además este nuevo controlador mejore en prestaciones al controlador en el que se basa, creemos que este objetivo se ha cumplido de forma exitosa.

- **Analizar el comportamiento y requerimientos de las aplicaciones que utilizan un gran volumen de datos usadas en servidores de gran potencia de cómputo, también conocidos como servidores HPC. Este estudio cubre la obtención de trazas de memoria y la extracción de estadísticas para distintas aplicaciones.** Como se puede observar en el capítulo 5.2 donde realizamos un análisis de las trazas para obtener más información, hemos cumplido este objetivo.

- **Analizar las prestaciones del controlador diseñado frente a un controlador de memoria NVRAM convencional.** De igual forma que el objetivo anterior, el resultado de este análisis se puede observar en la sección 5.1.

6.2 Relación con los estudios cursados

En general, este TFG aprovecha los conocimientos adquiridos de las asignaturas de *hardware* que se van estudiando a lo largo del grado, como pueden ser «Fundamentos de los computadores», «Arquitectura e Ingeniería de Computadores» o «Arquitecturas Avanzadas». Además, las siguientes competencias de la rama de «Ingeniería de Computadores» han resultado claves a la hora de la elaboración de este TFG.

- **Diseñar y construir sistemas digitales, incluyendo computadores, sistemas basado en microprocesador y sistemas de comunicaciones.** Hemos implementado una caché en un simulador reciente y ampliamente aceptado por la comunidad científica para acelerar las prestaciones de la memoria principal.
- **Analizar, evaluar, seleccionar y configurar plataformas hardware para el desarrollo y ejecución de aplicaciones y servicios informáticos.** En este sentido, hemos utilizado aplicaciones recientes utilizadas en los trabajos científicos. Estos *benchmarks* se han utilizado para obtener resultados de prestaciones y poder comparar distintos escenarios.
- **Analizar y evaluar arquitecturas de computadores, incluyendo plataformas paralelas y distribuidas, así como desarrollar y optimizar *software* para las mismas.** En esta memoria hemos realizado el análisis de prestaciones, así como evaluado las posibles ganancias que la modificación en la arquitectura del sistema simulado ha supuesto para las prestaciones del sistema global. No obstante, nos gustaría reseñar que el objetivo no era optimizar el diseño para maximizar las prestaciones sino familiarizarnos con estas herramientas complejas y adquirir ciertas habilidades necesarias para poder realizar trabajos posteriores en los próximos años.

6.3 Trabajo futuro

Como trabajo futuro de este TFG se prevé realizar distintas modificaciones para que la caché implementada sea más próxima a la de los procesadores comerciales. Por ejemplo, se implementarán y analizarán distintos algoritmos de reemplazo utilizados en las cachés de bajo nivel. Además de implementar estos cambios que afectarán al hit ratio como bien se comentaba en 5.1, se prevé modificar la estructura de caché. Esta cambio consistirá en convertirla en una caché por sectores, para averiguar el tamaño óptimo del sector realizaremos pruebas basándonos en el análisis teórico realizado en 5.2. Además de implementar una caché por sectores, se añadirá un mecanismo de prebúsqueda para ocultar todavía más la latencia de los módulos de memoria NVRAM y realizar una mejor explotación de la localidad espacial.

A parte de las modificaciones comentadas anteriormente, también podría ser interesante realizar otro estudio sobre el efecto del controlador de memoria con caché en sistemas cuya memoria principal esté implementado en DRAM, ya que estos podrían verse afectados de forma positiva. Aunque el beneficio que se espera sea menor, el poder aprovechar este controlador para recibir un aumento en prestaciones puede ser una forma de hacer frente al problema de la *memory wall*.

Este TFG sirve como primer paso en una cadena de estudios de organización de memoria principal híbrida, más concretamente, en el diseño de controlador, con tal de aprovechar las características de las memorias DRAM (rapidez) y NVRAM (alta capacidad de almacenamiento). Entre otros, el trabajo futuro se centrará en el papel que desarrolla la DRAM en la memoria híbrida. Analizaremos dos posibles opciones. O bien como una caché de la NVRAM y esta como memoria principal, o bien, el espacio de direccionamiento y la capacidad de memoria principal vienen dados por la suma de ambas. Nos centraremos en el aumento de las prestaciones de la caché. Para ello, se incorporará un mecanismo de prebúsqueda. Este mecanismo de prebúsqueda tiene ventajas e incovenientes, por una parte al prebuscar los datos en la caché se reduce la latencia. Por otra, se consume un mayor ancho de banda; es posible que cuando se necesite acceder el bus se encuentre ocupado realizando prefetch. Por tanto, los efectos del prefech solo se pueden conocer con simulación detallada. El primer paso es analizar el potencial de los bloques accedidos presentados en el capítulo anterior.

Asimismo, las propuestas recientes apuestan por el diseño del controlador de memoria híbrida en un chip aparte como el Intel© Optane™ e incorporar una caché SRAM para acelerar las prestaciones tanto de los módulos DRAM como NVRAM. En este sentido, en este TFG se ha puesto el primer granito de arena que pensamos continuar desarrollando en un futuro TFM y posterior tesis doctoral en los próximos años.

Bibliografía

[1] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 415–426. IEEE Computer Society, 2015.

[2] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2015.

[3] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[4] Onur Mutlu. Memory systems - lecture 1.2: Memory and dram basics (technion, summer 2018). https://www.youtube.com/watch?v=icyliEdf0x0. Último acceso 29 de junio de 2021.

[5] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379, 2012.

[6] Intel©. 3D Xpoint™ Technology revolutionizes storage memory. https://www.youtube.com/watch?v=Wgk4U4qVpNY. Último acceso 29 de junio de 2021.

[7] X-point memory cell - google patents. https://patents.google.com/patent/US6777705B2/en. Último acceso 29 de junio de 2021.

[8] Josué Feliu Pérez. Palnificació multinivell per a optimitzar l'ús de l'ample de banda de la jerarquia de memòria en multinuclis. TFG, Universitat Politècnica de València, 2011.

[9] Carlos Catalá Barber. Simulación de nuevas arquitecturas de memorias caché de procesadores para sistemas empotrados. TFG, Universitat Politècnica de València, 2011.

[10] Alejandro Valero Bresó. Reducció del consum energètic en memòries cache utilitzant cel·les estàtico-dinàmiques. TFG, Universitat Politècnica de València, 2009.

[11] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[12] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[14] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News*, 41:475, 07 2013.

[15] Peng gu, Benjamin Lim, Wenqin Huangfu, Krishna Malladi, Andrew Chang, and Yuan Xie. Nmtsim: Transaction-command based simulator for new memory technology devices. *IEEE Computer Architecture Letters*, PP:1–1, 05 2020.

[16] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.

[17] Matt Poremba and Yuan Xie. Nvmain: An architectural-level main memory simulator for emerging non-volatile memories. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 392–397, 2012.

[18] Bill Gervasi and Jonathan Hinkle. Overcoming system memory challenges with persistent memory and nvdimm-p. In *JEDEC Server Forum*, volume 2017, 2017.

[19] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro López. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.

[20] m5sim.com. Página de recursos de gem5. https://www.gem5.org/documentation/general_docs/gem5_resources/. Último acceso 28 de junio de 2021.

[21] m5sim.com. Modificación de la imagen de disco en simulación. http://www.m5sim.org/Disk_images#Modifications. Último acceso 29 de junio de 2021.

APÉNDICE A

Script para el lanzamiento de redis-benchamrk con 1000 peticiones por test

```
#!/bin/bash
redis-server &
sleep 5s
/sbin/m5 resetstats
/sbin/m5 checkpoint
redis-benchmark -n 1000 -t get,set,lpush,lpop
/sbin/m5 exit
```

APÉNDICE B

Script para el lanzamiento de MySQL-SysBench

```
    #!/bin/bash
    #preparacion del benchmark
    mysql -u root -e 'CREATE DATABASE test;'
    sysbench oltp_read_write --table-size=1000 --mysql-db=test --mysql-user=root
--db-driver=mysql prepare
    #inicio del benchmark
    /sbin/m5 resetstats
    /sbin/m5 checkpoint
    sysbench oltp_read_write --table-size=1000 --db-driver=mysql
--mysql-db=test --mysql-user=root --time=30 --max-requests=0 --threads=1 run
    /sbin/m5 exit
```

APÉNDICE C

Código del fichero de definiciones MySRAMCache.h

```
/*
* Esta clase es la representación del elemento "SRAM Cache Tags"
* del HMC
*/
#include "MyUIntStack.h"
#include <math.h>
#include <assert.h>

#include <string>
#include <vector>
#include <deque>
#include <iostream>
#include <list>
#include <iterator>
#include <unordered_map>
#include <bitset>

#define OUT

/*
*   Clase auxiliar que representa un byte de una línea de caché
*/
class MySRAMCacheEntry
{
  public:
    MySRAMCacheEntry( )
    {
      dato = 0;
      val[0] = 0;
```

```
    };

    MySRAMCacheEntry( uint8_t data )
    {
      dato = data;
      val[0] = 1;
    };

    ~MySRAMCacheEntry( )
    {

    };

    uint8_t getDato() const { return dato; }
    void setDato(const uint8_t &dato_) { dato = dato_; }

    void setVal() {val[0] = 1;}
    void resetVal() {val[0] = 0;}
    bool getVal() {return val[0];}

  private:
    uint8_t dato;
    std::bitset<1> val;

};

/*
*   Clase que implementa la funcionalidad de la caché
*/
class MySRAMCache
{

  typedef std::unordered_map<uint64_t, std::vector<MySRAMCacheEntry>> memoria_cache;
  public:
   /*
    * Crea una caché con máximo 2^n líneas (de 64 bits cada una)
    * y lat ciclos de latencia
    *
    * @param n exponente del tamaño máximo de memoria.
    * @param lat ciclos de latencia de la caché
    */
  public:
```

```
 MySRAMCache( uint64_t n, uint64_t lat ) : currSize(0)
 {
   maxSize = pow(2, n);
   latenciaCiclos = lat;
   srand(2021);
   dirArray.resize(maxSize);
 };
 ~MySRAMCache( );

/*
 * Comprueba si una dirección de memoria tiene datos válidos.
 *
 * @param addr Dirección a comprobar si tiene dato
 * @return TRUE si tiene un dato. FALSE si no.
 */
 bool hasData(uint64_t addr, uint64_t size);

/*
 * Devuelve la palabra que se encuentra en una dirección de memoria
 *
 *
 * @param addr Dirección de memoria a leer
 * @param size Numero de bytes a leer desde la dirección de memoria
 * @return Un array de uint8_t o nullptr si no existe una entrada en memoria.
 */
 uint8_t* readData(uint64_t addr, uint64_t size);

/*
 * Guarda una cantidad de bytes desde una dirección de memoria
 *
 * @param addr Dirección de memoria donde guardar el dato
 * @param data Dato a guardar (como uint64_t)
 * @param size Numero de bytes a guardar
 * @return TRUE si no ha habido problemas. FALSE si no.
 */
 bool writeData(uint64_t addr, uint8_t* data, uint64_t size);

/*
 * Invalida la palabra de una dirección de memoria
 *
 * @param addr Dirección cuya palabra se quiere invalidar.
 * @param size Numero de bytes a invalidar.
 * @return TRUE si no ha habido problemas. FALSE si no.
```

```
    */
    bool invalidateData(uint64_t addr, uint64_t size);

   /*
    * Getters y Setters
    */
    inline void setMaxSize(uint64_t maxSize_) {maxSize = maxSize_;}
    inline void setCurrSize(uint64_t currSize_) {currSize = currSize_;}

    inline uint64_t getMaxSize() {return maxSize;}
    inline uint64_t getCurrSize(){return currSize;}

    inline uint64_t getLatenciaCiclos(){return latenciaCiclos;}
  private:
    memoria_cache memoria;                        //Variable que almacena los datos
    const uint64_t mascaraDir =    0xFFFFFFFFFFFFFFC0;  //Variable para la etiqueta
    const uint64_t mascaraDesplz = 0x000000000000003F;  //Variable para el desplazamiento
    uint64_t maxSize, currSize;                   //Variables para gestionar el tamaño
    uint64_t latenciaCiclos;                      //Variable que contiene la latencia
    std::vector<uint64_t> dirArray;   //Vector para reemplazo RANDOM.
  protected:
};
```

APÉNDICE D

Código del fichero fuente MySRAMCache.cpp

```
#include "MySRAMCache.h"
#define TAMANYO 64

MySRAMCache::~MySRAMCache( )
{

}

bool MySRAMCache::hasData(uint64_t addr, uint64_t size)
{
  uint64_t moddedAddr = addr;
  uint64_t tag = moddedAddr & mascaraDir;
  uint64_t desplz = moddedAddr & mascaraDesplz;
  uint64_t porLeer = size;

  while(porLeer)
  {
    if(!memoria.count(tag))
      return false;

    porLeer = porLeer - (TAMANYO - desplz);
    if(porLeer < 1) //Asegurar el valor a 0 para finalizar el bucle
      porLeer = 0;

    moddedAddr += (TAMANYO - desplz);

    tag = moddedAddr & mascaraDir;
    desplz = moddedAddr & mascaraDesplz;
  }
```

```
  return true;
}

uint8_t* MySRAMCache::readData(uint64_t addr, uint64_t size)
{
  uint64_t moddedAddr = addr;
  uint64_t tag = moddedAddr & mascaraDir;
  uint64_t desplz = moddedAddr & mascaraDesplz;
  uint64_t porLeer = size;
  uint8_t *rv = new uint8_t[size];
  uint64_t posBaseRV = 0;
  std::vector<MySRAMCacheEntry> linea;

  while(porLeer)
  {
    //Leer datos de la dirección (alineada a TAMANYO Bytes)
    linea = memoria[tag];

    //Copiar los valores al vector de retorno
    int i = 0;
    for(;i + desplz < TAMANYO && porLeer; i++)
    {

      if(linea[i + desplz].getVal())
        rv[posBaseRV + i] = linea[i + desplz].getDato();
      else
        rv[posBaseRV + i] = 0;
      --porLeer;
    }
    posBaseRV = i;

    /*
    *   Actualizar la dirección añadiendo los bytes que hemos escrito + 1
    *   para indicar que queremos empezar a leer la primera palabra del
    *   siguiente bloque alineado con TAMANYO bytes.
    */
    moddedAddr +=  i;

    //Calcular la nueva etiqueta y el nuevo desplazamiento
    tag = moddedAddr & mascaraDir;
    desplz = moddedAddr & mascaraDesplz;
 }
```

```
  //Devolver el puntero al primer elemento (array) con los datos leidos.
  return rv;
}

bool MySRAMCache::writeData(uint64_t addr, uint8_t* data, uint64_t size)
{
  assert(data != 0x0);
  uint64_t moddedAddr = addr;
  uint64_t tag = moddedAddr & mascaraDir;
  uint64_t desplz = moddedAddr & mascaraDesplz;
  uint64_t porEscribir = size;
  uint64_t posBaseData = 0;


  while(porEscribir)
  {
    if(hasData(tag,porEscribir)) //Actualizar una linea ya existente
    {
      //Leer datos de la dirección (alineada a TAMANYO Bytes)
      std::vector<MySRAMCacheEntry>linea = memoria[tag];

      /*
      *   Insertar modificaciones hasta que no quepa más en la linea de cache
      *   (que corresponde a un bloque de TAMANYO Bytes)
      */
      int i = 0;
      for(; i + desplz < TAMANYO && porEscribir; ++i)
      {
        MySRAMCacheEntry datoAGuardar(data[i + posBaseData]);
        linea[i+desplz] = datoAGuardar;
        --porEscribir;
      }
      posBaseData = i;

      //Guardar la linea modificada
      memoria[tag] = linea;

      /*
      *   Actualizar la dirección añadiendo los bytes que hemos escrito + 1
      *   para indicar que queremos empezar a leer la primera palabra del
      *   siguiente bloque alineado con TAMANYO bytes.
      */
      moddedAddr +=  i;
```

```
  //Calcular la nueva etiqueta y el nuevo desplazamiento
  tag = moddedAddr & mascaraDir;
  desplz = moddedAddr & mascaraDesplz;
}
else //Añadir una linea nueva
{

  //Crear una línea nueva de TAMANYO entradas
  std::vector<MySRAMCacheEntry> wv(TAMANYO);

  //Añadir los datos a esta nueva línea
  int i = 0;
  for(; i < TAMANYO ; ++i)
  {
    wv[i] = MySRAMCacheEntry(0);
  }

  //Añadir los datos a esta nueva línea
  for(i = 0; i + desplz < TAMANYO && porEscribir ; ++i)
  {
    if(&wv + i + desplz != 0)
      wv[i+desplz] = MySRAMCacheEntry(data[i]);
    else
      wv[i+desplz] = MySRAMCacheEntry(0);
    --porEscribir;
  }

  /*
   * No cabría una linea nueva y por lo tanto
   * habría que sustituir una existente por la nueva
   */
  if(currSize + 1 > maxSize)
  {
    uint64_t indexToBeReplaced = rand() % maxSize; //[0, maxSize - 1]
    uint64_t tagToBeReplaced = dirArray[indexToBeReplaced];

    invalidateData(tagToBeReplaced,porEscribir);

    dirArray[indexToBeReplaced] = tag;

    assert(currSize < maxSize);
  }
```

```
      else
        dirArray[currSize] = tag;

      //Incrementar el tamaño
      ++currSize;

      //Guardar la línea nueva
      memoria[tag] = wv;

      /*
      *   Actualizar la dirección añadiendo los bytes que hemos escrito + 1
      *   para indicar que queremos empezar a leer la primera palabra del
      *   siguiente bloque alineado con TAMANYO bytes.
      */
      moddedAddr +=  i;

      //Calcular la nueva etiqueta y el nuevo desplazamiento
      tag = moddedAddr & mascaraDir;
      desplz = moddedAddr & mascaraDesplz;
    }
  }

  return true;
}

bool MySRAMCache::invalidateData(uint64_t addr, uint64_t size)
{
  uint64_t tag = addr & mascaraDir;

  if(!hasData(tag,size))
  {
    return false;
  }

  /* Eliminar de memoria*/
  memoria.erase(tag);
  --currSize;
  return true;

}
```

APÉNDICE E

Fichero de configuración de NVMain usados en la simulación

```
; Memory configuration file examples
; This configuration file is based on Samsung's 2012 ISSCC paper:
;     A 20nm 1.8V 8Gb PRAM with 40MB/s Program Bandwidth
;
; Phase Change Memory has a very high resistance ratio, allowing for
; global sense amplifiers. This configuration assumes global sense
; amplifiers, which decrease area at the cost of increased tRCD.
;
;

;===============================================================================
; Interface specifications

; 400 MHz clock (800 MT/s LPDDR). Clock period = 1.5 ns
CLK 667

; Data Rate. 1 for SDR, 2 for DDR
RATE 2

; Bus width in bits. JEDEC standard is 64-bits
BusWidth 64

; Number of bits provided by each device in a rank
; Number of devices is calculated using BusWidth / DeviceWidth.
DeviceWidth 8

; NVMain use CLK and CPUFreq to do the synchronization. The ratio CLK/CPUFreq
; is actually used. So a simple CLK=1 and CPUFreq=4 still works for simulation.
; However, it is straightforward to make it informative.
```

```
CPUFreq 2000
;===============================================================================

;*******************************************************************************
; General memory system configuration

; Number of banks per rank
BANKS 8

; Number of ranks per channel
RANKS 2

; Number of channels in the system
CHANNELS 2

; Number of rows in one bank
ROWS 65536

; Number of VISIBLE columns in one LOGIC bank
COLS 32 ; 32 horizontal tiles * 2048 bitlines / (8 device width * 8 burst cycles)

; Assume one large mat (no local sense amplifiers)
MATHeight 65536

; No refresh needed in PCM
UseRefresh false

; Not used in PCM, but we'll assign valid numbers anyway.
BanksPerRefresh 2
RefreshRows 4
DelayedRefreshThreshold 1

;*******************************************************************************

;===============================================================================
; Memory device timing parameters (in memory cycle)

tBURST 4 ; length of data burst

tCMD 1   ; Commands are 1 address bus cycle
tRAS 0   ; No row restoration needed
tRCD 48  ; 120ns @ 400 MHz = 48 cycles
tWP 60   ; Write pulse time. 150ns @ 400 MHz = 60 cycles
```

```
tRP 1    ; Precharge isn't needed. Writes occur only if needed
         ; and take tWP time during a precharge (write-back)
        ; or immediately (write-through)
tCAS 1   ; Assumes data is ready at the global sense amps after tRCD
tAL 0    ; 0 or 1
tCCD 2   ; usually 2 or 4, no more than tBURST

; The next set of timings is mainly based on control circuits, and the times
; are taken from normal LPDDR2 datasheets.
tCWD 4   ; 10ns
tWTR 3   ; 7.5ns
tWR 0    ; i.e., write-to-precharge, not needed here
tRTRS 1  ; for DDR-1, tRTRS can be 0
tRTP 3   ; No precharge, but still need to wait for data to leave internal
        ; FIFO buffers
tOST 0   ; No ODT circuitry in LPDDR

; These are mostly unknown at this point, but will likely
; be similar as they are meant to preserve power integrity
tRRDR 4
tRRDW 4
RAW 4
tRAW 20

; Powerdown entry and exit timings
tRDPDEN 5     ; Wait for read to complete - tCAS + tBURST
tWRPDEN 68    ; Wait for write to complete ... tAL + tCWD + tBURST + tWP
tWRAPDEN 68   ; No precharge, so same as tWRPDEN
tPD 1         ; Time from powerdown command to actually in powerdown mode
tXP 3         ; Time to power-up from power-down mode - 7.5ns
tXPDLL 200000 ; No DLL in LPDDR, will be used for deep power-down (tDPD) - 500us

; Refresh timings - not used in PCM, but we'll assign valid numbers anyway.
tRFC 100
tREFW 42666667
;=============================================================================

;*****************************************************************************
; Memory device energy and power parameters

; Read/write values are in nano Joules
; NOTES:
;   NVSIM energy is per word
```

```
;   Erd is the read energy from a single mat
;   Ewr is the write energy (SET or RESET, they are the same)
;   These values are the energys required to read a page into a row buffer.
;
;   Other energy values are taken from CACTI
;
EnergyModel energy
Erd 0.081200
Eopenrd 0.001616
Ewr 1.684811

; Subarray write energy per bit
Ewrpb = 0.000202

; Energy leaked in 1 sec (or just the wattage) in milli Joules
Eleak 3120.202

Epdpf 0
Epdps 0
Epda 0
Eref 0

; DRAM style power calculation. All values below in mA, taken from datasheet.

Voltage 1.5
;****************************************************************************

;============================================================================
; Memory controller parameters

; En nuestro caso, podemos usar cualquiera de los siguientes:
; PerfectMemory, FRFCFS o FRFCFS_CACHE
MEM_CTL PerfectMemory

; whether use close-page row buffer management policy?
; options:
;   0--Open-Page, the row will be closed until a row buffer miss occurs
;   1--Relaxed Close-Page, the row will be closed if no other row buffer hit
;   exists
;   2--Restricted Close-Page, the row will be closed immediately, no row
;      buffer hit can be exploited
ClosePage 0
```

```
; command scheduling scheme
; options: 0--fixed priority, 1--rank first round-robin,
; 2--bank first round-robin
ScheduleScheme 2

; address mapping scheme
; options: R:RK:BK:CH:C (R-row, C:column, BK:bank, RK:rank, CH:channel)
AddressMappingScheme SA:R:RK:BK:CH:C

; interconnect between controller and memory chips
; options: OffChipBus (for 2D), OnChipBus (for 3D)
INTERCONNECT OffChipBus

; FRFCFS-WQF specific parameters
ReadQueueSize 32 ; read queue size
WriteQueueSize 32 ; write queue size
HighWaterMark 32 ; write drain high watermark.
rite drain is triggerred if it is reached
LowWaterMark 16 ; write drain low watermark.
rite drain is stopped if it is reached
;===============================================================================

;*******************************************************************************
; Simulation control parameters
; Guardar traza de la ejecución ("PrintPreTrace" -> True), en el fichero
; "PreTraceFile"
;
; Tenemos que comentar "AddHook" y "PostTraceWritter" si usamos el controlador de
; memoria "PerfectMemory".
PrintGraphs false
PrintPreTrace true
PreTraceFile perfect-trace1k.trace
EchoPreTrace false
PeriodicStatsInterval 100000000

TraceReader NVMainTrace
;*******************************************************************************

;===============================================================================
; Endurance model parameters
; This is used for Non-volatile memory

EnduranceModel NullModel
```

```
EnduranceDist Normal
EnduranceDistMean 1000000
EnduranceDistVariance  100000
; Everything below this can be overridden for heterogeneous channels
;CONFIG_CHANNEL0 pcm_channel0.config
;CONFIG_CHANNEL1 pcm_channel1.config
; Set the memory is in powerdown mode at the beginning?
InitPD false
IgnoreData true
;================================================================================

; Descomentar estas líneas para obtener la traza en los controladores.
; Asegurarnos de que "PrintPreTrace" sea false.
; AddHook PostTrace
; PostTraceWriter NVMainTrace
```

APÉNDICE F

Script para la obtención de estadísticas de ocupación espacial a partir de una traza de NVMain.

```
#!/usr/bin/python3
import argparse
from typing import Optional

parser_arg = argparse.ArgumentParser(description='Obtiene estadísticas \
    de ocupación de los sectores a partir de un fichero de traza de NVMain\
    y del tamaño de bloque.')

parser_arg.add_argument('fichero_traza', metavar='f', type=str,
                        help='fichero de la traza a obtener\
                        las estdísticas de ocupación')

parser_arg.add_argument('--sectorSize', metavar='tamanyoEnByteDelSector',
                        type=int, help='tamaño del bloque a usar a la hora\
                        de interpretar la traza',default=256)

b_size = 64
args = parser_arg.parse_args()

#Leer fichero
f = open(args.fichero_traza)
traza = f.read()
f.close()

#Convertir la entrada en una lista
lineas_traza = str.splitlines(traza)
del lineas_traza[0] #La primera línea informa de la "Versión" de NVMain
```

```
tamanyoEnByteDelSector = args.sectorSize
#Definir variables
mascara_desplz_sector = tamanyoEnByteDelSector - 1
mascara_tag_sector = int('ffffffffffffffff', 16) - mascara_desplz_sector

#Depende del tamaño del bloque
mascara_desplz_bloque = b_size - 1
mascara_tag_bloque = int('ffffffffffffffff', 16) - mascara_desplz_bloque

bloques_por_sector = int(tamanyoEnByteDelSector/b_size)

sectores = {}

#Hacer MAP de cada acceso a su correspondiente página y bloque
for linea in lineas_traza:
    ciclo, op, dirMemStr, dato, datoAnt, idHilo, tamanyo = linea.split(' ')
    dirMem = int(dirMemStr, 16)

    etiquetaSector = mascara_tag_sector & dirMem
    etiquetaBloque = mascara_tag_bloque & dirMem

    #Comprobar si existe el sector en el dict
    if etiquetaSector in sectores.keys():
        #Al existir el sector, hay que comprobar si existe el bloque
        if etiquetaBloque in sectores[etiquetaSector].keys():
            #Si ya existe el bloque, solo hay que añadir uno
            sectores[etiquetaSector][etiquetaBloque] += 1
        else:
            #Como no existe el bloque, se crea la entrada con valor 1
            sectores[etiquetaSector][etiquetaBloque] = 1
    else:
        #Como no existe el sector, se puede añadir el bloque directamente
        sectores[etiquetaSector] = {}
        sectores[etiquetaSector][etiquetaBloque] = 1

#Reducir el map a los datos que queremos
resultados = [0] * int(round(tamanyoEnByteDelSector/b_size) + 1)
accesos_a_bloques = 0
for sector in sectores.keys():
    bloques_accedidos = len(sectores[sector])
    resultados[bloques_accedidos] += 1
```

```
#Mostrar resultados
print("Se han accedido a ", len(sectores),"sectores diferentes de tamaño",\
    tamanyoEnByteDelSector)
del resultados[0]
for index, resultado in enumerate(resultados):
    print("Sectores que han usado", index + 1, "bloque(s) =", resultado,\
        "(", round(resultado/len(sectores) * 100, 2) ,"% )")
```