UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Informática de Sistemas y Computadores

Mejora de las prestaciones en sistemas con memorias no volátiles

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Computadores y Redes

AUTOR/A: Avargues Gutiérrez, Miguel Antonio

Tutor/a: Sahuquillo Borrás, Julio

Cotutor/a: Petit Martí, Salvador Vicente

CURSO ACADÉMICO: 2021/2022

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Máster en Ingeniería de Computadores y Redes

Trabajo Fin de Máster

Mejora de las prestaciones en sistemas con memorias no volátiles

Autor: Miguel Antonio Avargues Gutiérrez

Tutores: Julio Sahuquillo Borrás,
Salvador Vicente Petit Martí

Valencia, Septiembre 2022

Índice general

Índice de figuras

Índice de tablas

Abstract

The current trend for cloud computing is largely due to the transfer of computational load from personal devices such as computers or mobile devices towards supercomputers or clusters. One of the most important issues in these cloud systems is the required high main memory capacities. This is a consequence of DRAM memory hitting a wall on integration density. To overcome this obstacle, there are alternative memory technologies able to replace DRAM. In particular, we focus on non-volatile random access memories - or NVRAM - with a greater integration density but worse access latency. In this context, there is a need to obtain similar performance to DRAM memory in order to NVRAM become a valid alternative. This work focuses on hiding the high latencies of NVRAM memories through the design of a two-level cache hierarchy at the memory controller level. We propose such a design and implement it on a simulated system to measure its impact on performance.

Keywords: Non-volatile memory; Main memory; Memory controller; NVRAM; NVMain; gem5

Resumen

La actual popularidad de la computación en la nube viene motivada por la migración de la carga computacional desde los dispositivos personales como ordenadores o dispositivos móviles a grandes supercomputadores o clústeres. Una de las mayores problemáticas que sufren estos sistemas de cómputo en la nube es la necesidad de grandes cantidades de memoria principal. Esto se debe a que la memoria DRAM se ha estancado en su densidad de integración. Para hacer frente a este hecho, existen varias tecnologías de memoria que pueden reemplazar a DRAM. En particular nos centramos en las memorias no volátiles de acceso aleatorio - o NVRAM - con una mayor escala de integración a cambio de peor latencia de acceso. Para que la NVRAM sea un reemplazo adecuado es necesario reducir su latencia con tal de que sea similar a la de la DRAM. Este TFM se centra en ocultar la alta latencia de las memorias NVRAM mediante el diseño de una jerarquía de caché de dos niveles a la altura del controlador de memoria. Para ello, se realizará un diseño y posterior implementación sobre un sistema simulado y se analizará su impacto en las prestaciones del sistema.

Palabras clave: Memoria no volátil; Memoria principal; Controlador de memoria; NVRAM; NVMain; gem5

Resum

L'actual popularitat de la computació en el núvol ve motivada per la migració de la càrrega computacional des dels dispositius personals com ordenadors o dispositius mòbils a grans supercomputadors o clústers. Una de les majors problemàtiques que sofreixen aquests sistemes de còmput en el núvol és la necessitat de grans quantitats de memòria principal. Això es deu al fet que la memòria DRAM s'ha estancat en la seva densitat d'integració. Per a fer front a aquest fet, existeixen diverses tecnologies de memòria que poden reemplaçar a DRAM. En particular ens centrem en les memòries no volàtils d'accés aleatori - o NVRAM - amb una major escala d'integració a canvi de pitjor latència d'accés. Perquè la NVRAM siga un reemplaçament adequat és necessari reduir la seva latència amb la condició que siga similar a la de la DRAM. Aquest TFM se centra en ocultar l'alta latència de les memòries NVRAM mitjançant el disseny d'una jerarquia de cache de dos nivells a l'altura del controlador de memòria. Per a això, es realitzarà un disseny i posterior implementació sobre un sistema simulat i s'analitzarà el seu impacte en les prestacions del sistema.

Paraules clau: Memòria no volàtil; Memòria principal; Controlador de memòria; NVRAM; NVMain; gem5

CAPÍTULO 1

Introducción

1.1 Descripción del problema

La computación actual se realiza en diferentes mercados o entornos. Dos de los más frecuentes en estos días son la computación móvil y la computación en la nube o *cloud computing*. Existe una simbiosis entre computación móvil y en la nube que se debe a las restricciones que sufren los dispositivos móviles. Aunque estos últimos han aumentado su potencia computacional a lo largo de los últimos años [1], especialmente desde la aparición de los primeros teléfonos inteligentes, siguen sufriendo limitaciones energéticas impuestas por su batería y por su capacidad de disipación.

Se han ideado diversas estrategias para mitigar los efectos de estas restricciones. Por ejemplo, el uso de componentes más eficientes, tecnologías de bajo consumo o técnicas de gestión dinámica de energía. Desafortunadamente, con frecuencia, estas técnicas repercuten de forma negativa en las prestaciones del dispositivo. Por ello, se plantea la necesidad trasladar cómputo desde los dispositivos móviles hasta otros más potentes en la nube. Esta estrategia no solo resulta interesante para los dispositivos móviles, sino que es una de las ideas principales tras el internet de las cosas (del inglés *Internet of Things*), también conocido simplemente IoT. De hecho, hoy en día es posible incluso el despliegue de clústeres para soportar servicios de este tipo.

Esta estrategia trae consigo sus propios retos ambientales y tecnológicos. Entre los cuales, en este TFM nos centramos concretamente en la memoria principal. Mover la mayor parte de la carga de trabajo a los servidores lleva a que estos necesiten alcanzar altas prestaciones para poder dar respuesta a todas las peticiones que se realizan. Para ello, las aplicaciones que se ejecutan en estos servidores necesitan una gran cantidad de da-

tos que se almacenan en la memoria principal. La cantidad de datos requerida continúa aumentando año tras año con las nuevas aplicaciones, por lo que la capacidad de almacenamiento de memoria principal se convierte en un factor clave para las prestaciones. Esto conlleva a un problema tecnológico ya que la memoria de acceso aleatorio dinámica - DRAM de sus siglas en inglés - sufre de problemas de escalabilidad.

Por este motivo, recientemente, se han introducido en el mercado tecnologías de memoria alternativas que proporcionan una mayor capacidad de almacenamiento. En este sentido, las tecnologías de memoria RAM no volátil (NVRAM) son una excelente alternativa ya que disponen de mucha mayor densidad de almacenamiento. Sin embargo, estas tecnologías presentan altas latencias de acceso por lo que es necesario idear mecanismos o técnicas para que estas memorias puedan utilizarse como memoria principal. Algunos trabajos proponen subsistemas de memoria híbrida [2, 3, 4, 5, 6, 7, 8] que constan de una fracción de módulos rápidos en tecnología DRAM y otra fracción de módulos de gran capacidad implementados con tecnología NVRAM, con el objetivo de combinar lo mejor de ambas tecnologías. Sin embargo, muchas de estas propuestas requieren de políticas de migración para decidir en tiempo de ejecución en qué tipo módulos deben almacenarse las páginas de memoria.

Otra aproximación, la seguida en este trabajo, es utilizar la memoria DRAM como cache de la NVRAM. En este sentido, en [9] se realizó una primera exploración de esta aproximación. En concreto, intentamos mejorar las prestaciones de un sistema con memoria no volátil añadiendo una memoria caché al controlador de memoria implementado en el simulador de memoria principal. Tras ello, se realizaron simulaciones de un sistema real con diferentes *benchmarks* para así cuantificar la mejora en prestaciones obtenida por la propuesta.

1.2 Motivación

Este trabajo muestra la evolución a lo largo del curso 2021-22 sobre la investigación realizada en la memoria principal tras la realización del TFG [9]. En este TFM nos centramos en la implementación de algunas ideas presentadas en el apartado «Trabajo Futuro» del citado TFG, los comentarios indicados por los miembros tribunal del TFG, y otras líneas de investigación que hemos creído interesantes desarrollar.

Otra de las motivaciones de este TFM ha sido profundizar más en herramientas ya utilizadas en el TFG, concretamente en los simuladores gem5 y NVMain, aplicar nuevas

herramientas, como CACTI, así como mejorar la metodología de investigación para aumentar la calidad de nuestra investigación en esta temática.

Finalmente, al continuar refinando las propuestas realizadas esperamos conseguir una mejora en prestaciones lo suficientemente significativa que justifique una implementación física de las técnicas desarrolladas.

1.3 Objetivos

El principal objetivo de este trabajo es que un sistema con memoria de acceso aleatorio no volátil - NVRAM de ahora en adelante - utilizada como memoria principal obtenga unas prestaciones similares a aquellas que obtendría el mismo sistema pero utilizando memoria DRAM. Para ello se plantean los siguientes hitos:

1. Diseñar una arquitectura de memoria que oculte las altas latencias de los módulos NVRAM.
2. Implementar la arquitectura de memoria diseñada en los simuladores del proyecto.
3. Realizar un análisis de las prestaciones del sistema con la nueva arquitectura.

1.4 Estructura de la memoria

El presente trabajo de fin de máster está dividido en los siguientes capítulos.

- Capítulo 2. Antecedentes. Este capítulo introduce las bases necesarias para la lectura y la comprensión de los temas abordados en el presente trabajo. El capítulo se divide en los dos campos principales de estudio. Por una parte se habla de conceptos relacionados con la arquitectura de computadores, centrándose en temas relacionados con los mecanismos de ocultación de la latencia de acceso a memoria, como son las memorias caché. Por otro lado se describen las memorias no volátiles y sus principales ventajas e inconvenientes frente a las memorias DRAM tradicionales.
- Capítulo 3. Estado del arte. Se introducen tecnologías de memoria no volatiles comerciales actuales, como Intel Optane y se presentan los principales desafios abordados en publicaciones científicas en lo que respecta a memorias principales híbridas y cachés DRAM.

- Capítulo 4. Entorno de trabajo para el desarrollo de los experimentos. Se presentan las características técnicas más relevantes sobre el entorno de pruebas que se empleará en el presente trabajo, así como las herramientas *software* que han hecho falta para poder desarrollar los objetivos de este proyecto.
- Capítulo 5. Propuesta. Se introduce el diseño del controlador de memoria híbrido, así como los componentes que lo componen y la relación entre ellos.
- Capítulo 6. Implementación. Se divide en dos secciones. La primera se dedica a la implementación de la caché de sectores en el simulador gem5. La segunda, introduce los cambios realizados en el código fuente del simulador NVMain para implementar la caché DRAM.
- Capítulo 7. Resultados experimentales. Este capítulo presenta y analizo los resultados obtenidos en los experimentos realizados en el entorno de pruebas.
- Capítulo 8. Conclusiones. Se presentan las principales conclusiones extraídas de los resultados del trabajo y las posibles ampliaciones que podrían derivar en trabajos futuros.

CAPÍTULO 2

Antecedentes

Este capítulo está dedicado a la descripción de los conceptos necesarios que facilitarán la lectura y comprensión del resto de los capítulos de este trabajo. Este capítulo se divide en dos secciones. En primer lugar, se estudian los temas relacionados con los mecanismos de ocultación de la latencia de acceso a memoria, en concreto la jerarquía de memoria cache. En segundo lugar, una introducción de las características principales de la tecnología de memoria no volátil.

2.1 Jerarquía de memoria

Los procesadores modernos incluyen memorias caché para ocultar la latencia de acceso a memoria principal acercando los datos a estas. Comparadas con memoria principal, las memorias caché presentan poca capacidad de almacenamiento, lo que permite que sean más rápidas. Las memorias caché se organizan en niveles, que, conjuntamente con la memoria principal, forman la jerarquía de memoria. Normalmente se implementan hasta tres niveles (*L1*,*L2*,*L3*), siendo el nivel superior (*L1*) de menor tamaño y con un tiempo de acceso menor, y, a medida que se baja de nivel (*L2*,*L3*) aumentan tanto la capacidad como el tiempo de acceso. La figura 2.1 muestra un ejemplo de jerarquía de memoria con tres niveles de caché.

El funcionamiento de esta jerarquía de memoria es el siguiente. Cuando el procesador ejecuta una instrucción para acceder a datos en memoria, se hace una búsqueda por todos los niveles de la jerarquía empezando por el más cercano al procesador. Si se encuentran los datos en este primer nivel (*L1*), se produce un acierto (*hit* en inglés). Por otro lado, si no se encuentran los datos se produce un fallo (*miss*), y por tanto se busca el dato en

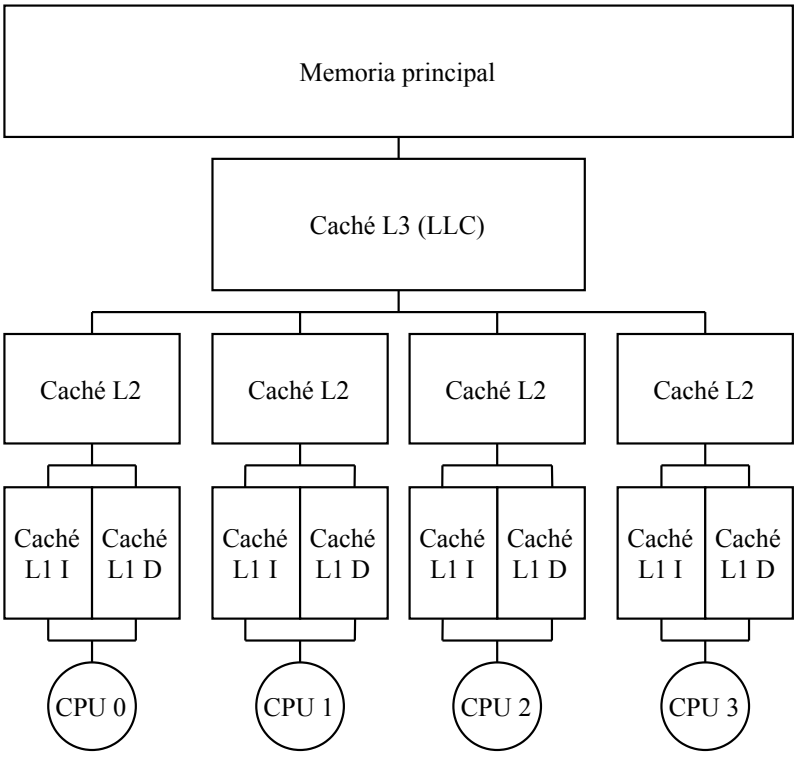

Figura 2.1: Ejemplo de una jerarquía de memoria de tres niveles.

nivel de caché contiguo más alejado del procesador. En el caso de no existir un siguiente nivel de caché, se accede a la memoria principal, lo que supone un tiempo de acceso muy elevado que repercute negativamente en las prestaciones.

Para evitar los accesos a memoria principal y conseguir buenas prestaciones, en los últimos años se ha aumentando la capacidad de la memoria caché del nivel más alejado del procesador: *LLC* (*Last Level Cache*). Esta caché almacena decenas de megabytes y tiene como objetivo aumentar la tasa de aciertos y evitar, en la medida de lo posible, los accesos a memoria principal. Normalmente la LLC se comparte entre todos los núcleos del procesador para así tener un mayor aprovechamiento en el caso de que los núcleos vecinos no hagan uso de su espacio, pudiendo utilizarlo otro núcleo que sí lo necesite. A raíz de esto, surge el problema de que se generen interferencias entre núcleos y de esta forma haya un impacto negativo en las prestaciones de las aplicaciones que se estén ejecutando concurrentemente.

2.2 Tecnologías de memoria no volátil

Como se ha comentado en la sección 1.1, la escalabilidad de las memorias DRAM se ha ralentizado en los últimos años. Desde 1985, esta tecnología había multiplicado su capacidad a un ritmo de dos veces cada año y medio. Sin embargo, desde principios de este siglo esta tendencia ha ido disminuyendo para tamaños de nodo menores que 10 nm por problemas de fiabilidad [10].

Es por ello que desde hace algunos años mucha investigación se ha centrado en el desarrollo de otras tecnologías alternativas a la DRAM. Entre ellas, cabe destacar las tecnologías denominadas *no volátiles*. Estas tecnologías proporcionan almacenamiento persistente aún sin una alimentación eléctrica continua. Algunas de las tecnologías más estudiadas son: las memorias de acceso aleatorio magneto-resistivas (MRAM), las memorias de cambio de fase (*phase-change memory* en inglés, o simplemente PCM), o las memorias de acceso aleatorio resistivas (RRAM o ReRAM).

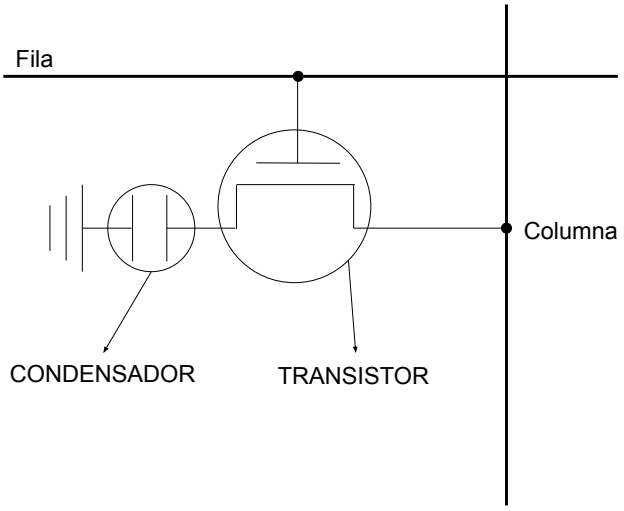

Figura 2.2: Celda de memoria DRAM.

Estas memorias cambian la arquitectura del nodo de almacenamiento de memoria. En DRAM, como se puede observar en la figura 2.2, este nodo está formado por un transistor y un condensador que se conectan a la columnas de selección y a la fila de bits. En contraposición, y a modo de ejemplo, se muestra en la figura 2.3 la arquitectura de una memoria Intel© Optane™ 3D-Xpoint™. Esta está formada por un selector (bloque amarillo) y un elemento de memoria (bloque verde). Este par de elementos se rodea de dos conductores (prismas rectangulares grises oscuros) por el lado superior e inferior,

que sirven para direccionar de forma única a una sola celda. Este nodo puede replicarse fácilmente en todas las direcciones, incluso se puede apilar, lo que incrementa mucho la densidad de memoria de este tipo de celdas. Además, gracias a la naturaleza no volátil de las celdas de memoria NVRAM no es necesario el refresco, tal como ocurre en las memorias DRAM debido a que los condensadores pierden carga a lo largo del tiempo.

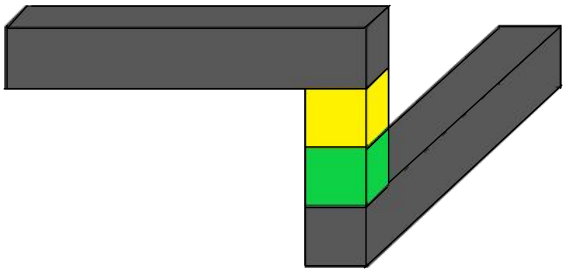

Figura 2.3: Celda de memoria usada en las memorias OPTANE.

Lamentablemente, como ya se ha introducido anteriormente, no todo son ventajas, ya que las memorias no volátiles suelen ser unas 3 veces más lentas que las DRAM. Esto reduce su viabilidad como reemplazo a la memoria DRAM en los sistemas actuales de forma sensible, ya que la diferencia de velocidades incurre de forma negativa en las prestaciones del sistema. Esto se debe a que que el subsistema de memoria tiene un gran peso en las prestaciones globales al alimentar con datos e instrucciones a la CPU, que es unas 1000 veces más rápida, por lo que suele ser el principal cuello de botella de los sistemas actuales.

Por otro lado, además de las latencias que pueden resultar prohibitivas, las memorias no volátiles soportan un número limitado de operaciones de escritura. Por lo tanto, el mero hecho de realizar escrituras a lo largo de la vida del sistema en el que se utilizan degrada sus prestaciones hasta provocar su mal funcionamiento. Todos estos problemas conllevan la necesidad de una solución que oculte las altas latencias de acceso de estas memorias que, al mismo tiempo, mejore su durabilidad.

CAPÍTULO 3

Estado del arte

En este trabajo se presenta el actual estado del arte relacionado con la temática del presente trabajo. Primero, se introducen la nueva tecnología de memorias no volátiles Intel Optane. Posteriormente, se presenta el concepto de memoria principal híbrida, compuesta por dos o más tecnologías de memoria. Finalmente, se mencionan los desafíos de diseño de las cachés DRAM.

3.1 Intel Optane

Debido al incremento en la necesidad de datos que exhiben las aplicaciones modernas como bases de datos en memoria, análisis y procesamiento de grafos o sistemas de aprendizaje automático, sus prestaciones se benefician de aumentar la capacidad de memoria principal. Por ello, Intel lanzó al mercado «Optane» [11].

Los *DIMM* Optane ofrecen una gran capacidad de almacenamiento y se pueden conectar junto a los DIMM DRAM convencionales, en configuraciones de memoria principal híbridas. Sin embargo, existen diferencias sustanciales entre las tecnologías de memoria que implementan ambos tipos de DIMM. Por ejemplo, la memoria no volátil de los DIMM Optane presenta anchos de banda asimétricos de lectura y escritura y granularidad de acceso a los bancos de memoria mayor que una línea de cache típica (64 B). Estas diferencias, hacen de Optane una memoria mucho más sensible al patrón de accesos de las aplicaciones que se estén ejecutando. Además, las limitaciones de ancho de banda resultan en un serio problema de escalabilidad al realizarse accesos de forma paralela. Cuando una aplicación consta de cuatro o más hilos realizando accesos a memoria, la limitación en el ancho de banda de las escrituras tanto secuenciales como aleatorias causa conten-

ción en el acceso a memoria que degrada las prestaciones. A esta degradación se suma la causada por las altas latencias de escritura de la NVRAM, mucho mayores que las de DRAM (entre 10,7 y 16,5 veces, para escrituras secuenciales y aleatorias respectivamente) [12].

Por otro lado, el gran aumento de la capacidad de la memoria principal se convierte en un inconveniente a nivel de gestión de memoria del sistema operativo. Normalmente las decisiones de la política de gestión de memoria se realizan iterando sobre la tabla de páginas y leyendo la información que contiene. La sobrecarga por acceder a esta tabla incrementa al aumentar la capacidad de memoria. En sistemas con cientos de gigabytes el tiempo de buscar la tabla es del orden de décimas de segundo. Para sistemas con varios terabytes de almacenamiento, el tiempo de búsqueda pasa a ser de varios segundos. Esta tendencia causa que la gestión de memoria sea insostenible [12].

3.2 Memoria Híbrida

Con tal de hacer frente a estos inconvenientes, la investigación reciente [2, 3, 4, 5, 6, 7, 8] plantea una estructura de memoria principal en la que coexisten dos tipos diferentes de memoria en canales diferentes. Usualmente estos dos tipos de memoria se complementan el uno al otro, una memoria pequeña pero rápida (memoria A) y una memoria más grande pero considerablemente más lenta (memoria B).

En esta organización de memoria híbrida, el espacio de direccionamiento total del sistema es la suma del canal con memoria de tipo A (canal A) y el canal con memoria del tipo B (canal B). Aunque el espacio de direccionamiento puede ser común, para las aplicaciones no siempre es posible acceder a ambos tipos de memoria indistintamente. Por ejemplo, ciertas aplicaciones que dependen de un flujo constante de gran cantidad de datos se ven muy perjudicadas al utilizar tecnologías o canales de memoria con poco ancho de banda, llegando incluso a dejar de funcionar correctamente.

La gestión de estas organizaciones heterogéneas de memoria principal puede realizarse de varias maneras: a nivel software, bien mediante librerías o mediante el sistema operativo (de forma transparente a las aplicaciones); o a nivel hardware, mediante el controlador de memoria que maneja todo el espacio de direccionamiento y la lógica encargada de seleccionar dónde almacenar cada dato.

La figura 3.1 representa un ejemplo de un sistema con arquitectura de memoria híbrida. Como se puede apreciar en la figura, a través de el controlador o controladores de memoria del sistema se accede a los datos no disponibles en la LLC. Esos datos pueden ubicarse en la Memoria A o en la Memoria B, que se comunican con el procesador a través del canal correspondiente.

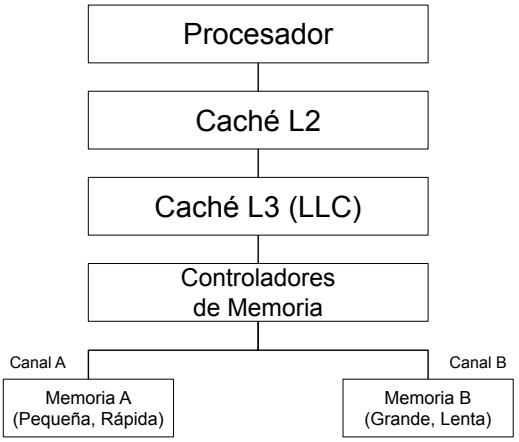

Figura 3.1: Diagrama de una arquitectura de memoria híbrida.

3.3 Cachés DRAM

Mediante la tecnología DRAM se pueden llegar a implementar memorias caché de gran tamaño (del orden de cientos de megabytes o incluso de unos pocos gigabytes). Es por ello que en este TFM se plantea su uso para mejorar las prestaciones en subsistemas de memoria que incorporan tecnologías de memoria no volátiles. Sin embargo, para mejorar las prestaciones de sistemas con cachés tan grandes es necesario hacer frente a algunos desafíos clave:

- Organizar el almacenamiento de las etiquetas.
- Optimizar la latencia en aciertos.
- Tratar los fallos de forma eficiente.

Estos desafíos ya se plantearon en el diseño de las LLC en procesadores comerciales [13], por lo que cabría esperar que alguna de las soluciones propuestas en ese contexto sean aplicables a las cachés DRAM. Sin embargo, esto no es necesariamente cierto debido a que la tecnología DRAM es más lenta que la de las cachés tradicionales [14]. Por lo tanto, las aproximaciones que degradan la latencia de acierto de forma significativa para incre-

mentar la tasa de aciertos pueden hacer que la latencia media de acceso (que tiene en cuenta la tasa de aciertos, latencia de acierto y latencia de fallo) empeore.

Por esta razón, una de las prioridades de diseño de las cachés DRAM es controlar la latencia de acierto. En este sentido, las aproximaciones del estado del arte [14, 15, 16, 4, 17, 18] proponen diversos mecanismos para evitar incrementarla o incluso reducirla. Otros objetivos de diseño a tener en cuenta son reducir la latencia en caso de fallo y aumentar la tasa de aciertos.

Algunos diseños de cachés DRAM se basan en almacenar por separado etiquetas y datos (cachés divididas). Las etiquetas se almacenan en una memoria SRAM más rápida, que indica, para cada acceso, si el dato accedido se encuentra en la memoria DRAM y, en caso afirmativo, la dirección en la que se encuentra. Por esto, cada acceso a la caché DRAM implica un acceso a la memoria SRAM de etiquetas, lo que puede afectar la latencia de acierto. En cachés del tamaño de decenas de megabytes esta latencia no es muy grande y es asumible. Sin embargo, cuando se trata de una caché DRAM con una capacidad similar o mayor que cientos de megabytes, la latencia para acceder las etiquetas comienza a resultar prohibitiva, haciendo necesario el uso de otras organizaciones de la memoria.

CAPÍTULO 4

Entorno de trabajo

Este capítulo describe la plataforma de pruebas utilizada para el desarrollo de los experimentos y sus características más relevantes. También se presenta el *software* empleado, distinguiendo entre simuladores y otras herramientas.

4.1 Equipo de pruebas

En la tabla 4.1 se describen las características principales del sistema sobre el que se ha desarrollado todo el trabajo descrito en este TFM.

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	2x32 GB DDR4-2400 DRAM
Sistema Operativo	Ubuntu 20.04.4 LTS
Kernel	5.14.0-1044-oem

Tabla 4.1: Características del sistema utilizado para el lanzamiento de simulaciones.

4.2 Cargas experimentales

Hemos usado distintas cargas experimentales - o *benchmarks* - representativas de aplicaciones reales que se ejecutan en servidores, para estudiar el comportamiento del sistema y poder medir sus prestaciones. Las aplicaciones o *benchmarks* que usaremos para la evaluación de la propuesta son Redis, MySQL y Memtier.

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. Redis dispone de un programa opcional conocido como «redis-benchmark»,[1] que se ocupa de cargar el servidor a base de peticiones para evaluar el rendimiento del 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.

Memtier_benchmark - o simplemente **Memtier** - es una herramienta capaz de generar diversos patrones de tráfico de peticiones a instancias tanto de servidores Redis, como de servidores Memcached. Es una utilidad por línea de comandos capaz de lanzar varios hilos cada uno controlando una cantidad configurable de clientes, controlar la tasa entre peticiones de tipo «get» y «set» y producir informes de ejecución del *benchmark*. Como ya se dispone de una instancia de un servidor Redis creemos que resulta de interés el uso de una carga de trabajo alternativa con otros patrones de acceso.

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 resultados experimentales usamos SysBench, el cuál está pensado para realizar pruebas sobre sistemas con bases de datos MySQL, entre otros.

[1] Cuando hablamos del *benchmark* «Redis», nos referimos a «redis-benchmark».

4.3 Simuladores

Debido a la complejidad y coste de realizar cambios sobre un sistema real, es frecuente el uso de simuladores con tal de implementar y probar diferentes lineas de investigación. Gracias a estas herramientas, se puede obtener una estimación que las modificaciones introducidas tendrían en un sistema real.

En este trabajo, se utilizan dos simuladores diferentes. NVMain, encargado de simular la memoria principal, y gem5, encargado de simular el resto del sistema (procesador, caches, E/S y memoria secundaria). A continuación se introducen de forma más detallada ambos simuladores.

4.3.1. gem5

El simulador gem5 [19] es un simulador que funciona mediante eventos y es capaz de simular todo un sistema completo, desde la unidad central de procesamiento hasta la entrada y salida. Es por tanto un simulador muy rico en características pero a su vez muy complejo, ya que abarca muchos componentes y la relación entre ellos. El hecho de que funcione mediante eventos, permite una simulación rápida del sistema que recrea. Sin embargo el sistema de coherencia de mensajes con los que se comunican los diferentes elementos - conocidos como *SimObjects* - añade una gran dificultad para implementar componentes personalizados. Se pueden modificar gran cantidad de los parámetros de cada *SimObject* en los ficheros de configuración encargados de preparar el sistema simulado.

Este permite dos modos de simulación: sistema completo (modo *full-system* o **FS**) y emulación de llamadas al sistema (*syscall emulation* o **SE**). El primer modo es mucho más realista pero también resulta más lento al simular todos los componentes software y hardware de un sistema real tales como sistema operativo, dispositivos de entrada/salida, etc. El segundo modo es menos preciso pero su simulación resulta más rápida. En la mayoría de casos el modo SE es el recomendado debido a que requiere de menos preparación y no implica la instalación de un sistema operativo. Sin embargo, para este trabajo se va a utilizar el modo FS ya que buscamos replicar un sistema de la manera más realista posible, incluyendo las interferencias del sistema operativo y su impacto en las prestaciones.

Aunque capaz de simular todo el sistema, gem5 ofrece la opción de compilarse conjuntamente con otros simuladores, trabajando de forma coordinada para mejorar el realismo

de la simulación. Como este trabajo se centra en las memorias no volátiles utilizaremos un entorno de simulación híbrido compuesto por gem5 y el simulador de memorias NVMain.

4.3.2. NVMain

Pese a que gem5 es capaz de simular memoria principal, en este trabajo hemos utilizado NVMain [20] para esta función. Este simulador preciso a nivel de memoria principal es capaz de emular el funcionamiento de diversos tipos de tecnologías no volátiles, así como de otras más convencionales como DRAM. Además, ofrece la posibilidad de simular organizaciones de memoria principal compuestas de diferentes tecnologías o híbridas.

NVMain se puede ejecutar de forma autónoma, en cuyo caso funciona mediante las trazas que se le ofrecen como entrada o puede utilizarse como librería en otros simuladores (gem5 en nuestro caso) como modelo de memoria principal. De forma similar a gem5, los ficheros de configuración de NVMain ofrecen una forma de ajustar los diferentes parámetros temporales, espaciales y de consumo de las memorias modeladas.

4.3.3. Instalación de los simuladores

En este apartado se definen los pasos para compilar ambos simuladores conjuntamente, indicando las dependencias y versiones de software necesarios para que el proceso se complete de forma satisfactoria.

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 Linux 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 20.04 ejecutamos la siguiente orden:

```
1 sudo apt install build-essential git m4 scons zlib1g zlib1g-dev
  ↪ libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev
  ↪ python3-dev python-is-python3 libboost-all-dev pkg-config gcc-7 g++-7
  ↪ pip
```

Para facilitar la futura compilación con NVMain, recomendamos también instalar (o hacer *downgrade* si ya están instalados en una versión más reciente) 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 3.1.2, 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/3.1.2/scons-3.1.2.tar.gz
tar -xzvf scons-3.1.2.tar.gz
cd scons-3.1.2
python setup.py install #Es posible que sea necesario el uso de sudo
```

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-3.1.2
```

Una vez hemos realizado la instalación de SCons, podemos proceder a clonar el repositorio de gem5 versión 21.0.1.0 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 v21.0.1.0
```

Tras obtener los ficheros necesarios desde el repositorio, debemos instalar el paquete «six». Esto se puede conseguir con el siguiente conjunto de órdenes:

```
cd gem5
pip install six
```

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».

Cabe destacar que no tiene que ser necesariamente el binario «gem5.opt» el objetivo de la compilación, ya que existen diferentes objetivos como «gem5.debug», el propio «gem5.opt» o «gem5.fast», que puede acelerar algunas simulaciones. Finalmente, el parámetro «-j $(nproc)» especifica la cantidad de *threads* a usar en el momento de la compilación. Recomendamos usar la máxima cantidad posible de hilos ya que la compilación se acelera enormemente gracias a esto. En caso de querer dedicar una cantidad menor que los hilos que soporta el sistema, se puede sustituir «$(nproc)» por la cantidad de hilos deseada.

NVMain

NVMain tiene escasas dependencias y todas se comparten con gem5; por ello, las instalamos de forma conjunta en la sección 4.3.3. En cualquier caso, conviene recordar las versiones necesarias para el funcionamiento de este.

- GCC y G++: versión 7.
- Python: versión 3.8.10; es posible que funcione con versiones más antiguas si son de Python 3.X. El uso de cualquier versión de Python 2 puede llevar a que sea imposible compilar el binario.
- SCons: versión 3.1.2.

Para obtener los ficheros fuente de NVMain debemos clonar el repositorio que contiene uno de los parches necesarios preaplicado para la compilación conjunta con gem5. Este

no tendrá efecto en la configuración en solitario. Para obtener este repositorio ejecutamos la siguiente orden:

```
1 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 en solitario (es decir, sin gem5), la cual se discute en el próximo apartado. Para ello, ejecutamos la siguiente orden en el directorio base de NVMain:

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

De modo similar a gem5, NVMain dispone de diferentes objetivos posibles para 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, podemos solucionar este error simplemente comentando esta línea con una almohadilla «#». Una vez comentada, la compilación del binario de NVMain funcionará sin problemas.

Compilación conjunta

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 conjunta. 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 los experimentos y obtener los resultados.

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[2] añadiremos el siguiente código respetando las tabulaciones de Python y dejando una línea en blanco tras realizar el cambio:

[2]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.

```
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")
```

También es necesario modificar el fichero «NVMainMemory.py» ubicado en «NVmain/-Simulators/gem5», hay que sustituir las líneas que hacen referencia a «AbstractMemory» y «ClockDomain» justo antes de definir la clase «NVMainMemory» por:

```
from m5.objects.AbstractMemory import *
from m5.objects.ClockDomain import *
```

Así mismo, es necesario actualizar el objeto de memoria que que define NVMain con gem5 con tal de adaptarse a la versión 21.0.1.0. Para ello es necesario sustituir todo el contenido de «nvmain_mem.hh» ubiacado en «NVmain/Simulators/gem5» por el contenido del anexo A.

El fichero «nvmain_mem.cc» con todos los cambios necesarios se puede obtener en el anexo B.

Una vez realizadas estas modificaciones, 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:

```
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[3] con todos los ficheros necesarios 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 como su sistema de memoria. Estos argumentos son:

- cpu-type: Este argumento debe modificarse debido a que NVMain solo funciona correctamente cuando gem5 envía paquetes de tipo *timing*. Sin entrar en detalles sobre los tipos de paquetes de gem5, indicar que la CPU debe ser capaz de enviar paquetes en modo *timing*. Un tipo de CPU incluido en gem5 que es capaz de ello es TimingSimpleCPU.

[3] https://github.com/miavgu/nvram-tfm. Última visita 12/09/2022.

- mem-type: En este argumento se debe indicar a gem5 que queremos que NVMain modele la memoria principal. Es por ello que el valor de este parámetro debe ser «NVMainMemory».

- nvmain-config: Este argumento se incluye con las modificaciones que permiten la ejecución conjunta de ambos simuladores. NVMain necesita un fichero de configuración para poder funcionar, y este argumento indica cuál es este fichero. El valor debe ser el *path* hasta un fichero de configuración de NVMain.

Por ejemplo, una orden para poder usar el simulador 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
```

4.4 Otras herramientas

4.4.1. CACTI

CACTI 7.0 (una evolución de CACTI 6.0 [21]) es una herramienta desarrollada por Hewlett Packard capaz de modelar diversos tipos de memorias con tal de obtener una estimación de su latencia de acceso y consumo. Para ello, hay que seleccionar diferentes atributos sobre la memoria a modelar, como pueden ser el número de puertos, la organización, o el nodo de la tecnología que implementa la memoria. Los atributos seleccionados se incluyen en un fichero de configuración.

El uso de esta herramienta permite modelar de forma realista las latencias de las memorias caché. En este TFM se han desarrollado tantos ficheros de configuración como memorias caché implementa el sistema simulado[4].

[4]Las memorias caché en el sistema simulado se pueden consultar en 7.1.

Obtención

CACTI se puede obtener desde un repositorio público[5] que contiene todo el código fuente necesario. Para poder utilizar la herramienta, es necesario primero obtener los fuentes para la compilación, lo que se puede realizar con la orden:

```
1 git clone ttps://github.com/HewlettPackard/cacti
```

Tras la descarga, se puede compilar ejecutando la orden *make* en el directorio:

```
1 cd cacti
2 make
```

Una vez el proceso de compilación se haya completado, un binario llamado «cacti» debería haberse creado en el directorio.

Uso

Como se ha mencionado al principio de esta sección, CACTI necesita un fichero de configuración para estimar los valores de latencia y consumo. El directorio contiene un fichero de ejemplo denominado «cache.cfg». En él, se pueden observar una gran cantidad de variables relacionadas con memorias principales y memorias caché. Las más relevantes para este proyecto son:

- Tamaño - *size*
- Tamaño de línea de caché - *block size*
- Asociatividad - *associativity*
- Tamaño de la tecnología utilizada (en micrómetros) - *technology (u)*
- Puertos de lectura y escritura (tanto compartidos como exclusivos) - *read-write port / exclusive read port / exclusive write port*
- Tipo de memoria que implementará la memoria o caché a modelar - *data array cell type / data array peripheral type / tag array cell type / tag array peripheral type*
- Tipo de memoria a modelar (Caché, Memoria Principal, RAM) - *cache type*

[5] https://github.com/HewlettPackard/cacti. También forma parte del repositorio del TFM https://github.com/miavgu/nvram-tfm. Última visita 12/09/2022.

En el anexo C se puede observar, a modo de referencia, el fichero de configuración utilizado para obtener los parámetros temporales para la caché de nivel dos del sistema. También están disponibles en el repositorio del proyecto.

CAPÍTULO 5

Propuesta

Para cumplir con los objetivos presentados en el apartado 1.3 del presente proyecto nos planteamos:

- Mejorar el controlador de memoria con una memoria caché de sectores.
- Implementar una organización de memoria híbrida compuesta por memoria principal no volátil (NVRAM) y una caché dividida. Los datos de la caché dividida se almacenarán en una estructura DRAM. La caché dividida también puede ser referida como «caché DRAM».

Cada una de estas mejoras ayuda a ocultar la latencia de la memoria NVRAM. Con tal de obtener una latencia de acceso similar (o inferior) a la tecnología DRAM, es necesario que la memoria DRAM que almacena los datos de la caché dividida sea más rápida que una memoria DRAM trabajando como memoria principal. Una de las formas más sencillas de obtener este tipo de latencia es reduciendo el tamaño de la memoria DRAM. Para una caché a nivel de controlador de memoria será necesaria una memoria del orden de decenas de MB, no de GB. Esta drástica reducción en tamaño permite implementar memorias DRAM bastante más rápidas que las convencionales.

Otra ventaja que tiene esta propuesta es la de reducir el número de lecturas y escrituras que se realizan sobre la memoria NVRAM, lo que reduce el desgaste y alarga la vida útil de esta memoria. La reducción en el número de operaciones a memoria se debe a que los dos niveles de caché filtran muchos de los accesos, evitando que sean atendidos por la NVRAM.

5.1 Componentes

El diseño del controlador de memoria consta de dos pasos:

1. Definir los componentes individuales que componen el controlador de memoria
2. Especificar la funcionalidad de cada componente, así como la interacción con el resto de ellos.

La implementación que se propone en el capítulo 6 requiere de cuatro componentes principales:

- **Caché de sectores**. Implementada en el chip del controlador de memoria, agrupa bloques consecutivos de datos en sectores. Se plantea una caché de sectores con el objetivo aprovechar la localidad espacial y así reducir la latencia media al acceder a memoria NVRAM. Esto se consigue trayendo varios bloques consecutivos (1 sector) cuando se realiza un acceso a NVRAM. Por su diseño, una caché de sectores actúa como un *prefetcher* secuencial. La caché de sectores contiene bits de control sobre el estado de los sectores y de cada bloque individual. Esta caché es la primera estructura que se comprueba cuando un acceso desde el último nivel de la jerarquía de caché del procesador llega al controlador de memoria.
- **Dispositivos DRAM**. Dispositivos DRAM DIMM convencionales que se utilizan para proporcionar un segundo nivel de almacenamiento para ayudar a reducir aún más el tiempo de acceso medio a NVRAM. El objetivo es que una memoria DRAM más rápida (pero más pequeña) sea la que se accede, minimizando el uso de la memoria con tecnología NVRAM.
- **Caché de etiquetas**. Junto con los dispositivos DRAM - que almacenan los datos - forma una **Caché DRAM** o **Caché dividida**. Acceder directamente a DRAM por cada fallo en la caché de sectores resultaría en un gran sobrecarga por accesos infructuosos, ya que el bloque podría estar almacenado en NVRAM. Para evitar este problema, se implementa una caché de etiquetas en SRAM que contiene las etiquetas de los bloques almacenados en DRAM y evita accesos inútiles a esta última. Una implementación en SRAM proporciona un acceso rápido a las etiquetas y se ha usado en procesadores modernos. Por ejemplo, el IBM POWER4 [13] implementa la caché de tercer nivel en un chip diferente al del procesador mientras que las etiquetas se almacenan en el procesador para evitar una excesiva penalización en la latencia.

- **Dispositivos NVRAM**. Contiene los DIMM de tecnología NVRAM y se utilizan como memoria principal del sistema.

La figura 5.1 muestra el controlador de memoria propuesto, incluyendo los componentes descritos y el flujo de datos entre ellos.

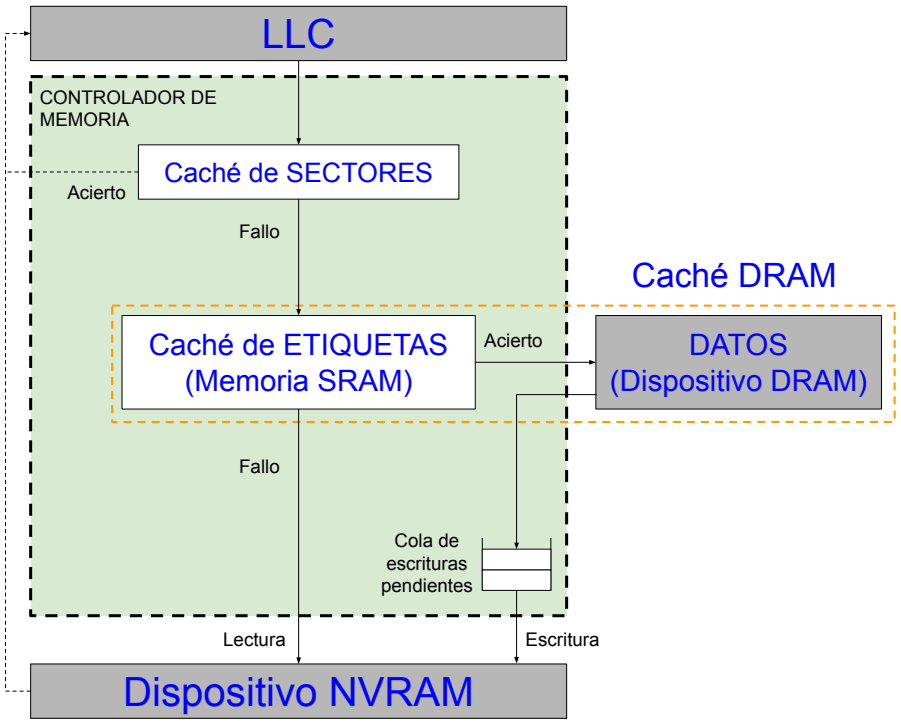

Figura 5.1: Componentes del controlador de memoria y flujo de datos entre ellos.

CAPÍTULO 6

Implementación

La implementación de ambas cachés se trata por separado ya que cada una de ellas se implementa en un simulador diferente. El principal motivo de esta separación es que los datos de la caché DRAM se almacenan en NVMain (simulador de los DIMM de memoria), por lo que almacenar las etiquetas en gem5 complica la implementación. Por otro lado, gem5 soporta de forma nativa cachés de sectores, por lo que su implementación resulta más sencilla en gem5.

La relación entre componentes simulados y dónde se implementan puede observarse en la figura 6.1.

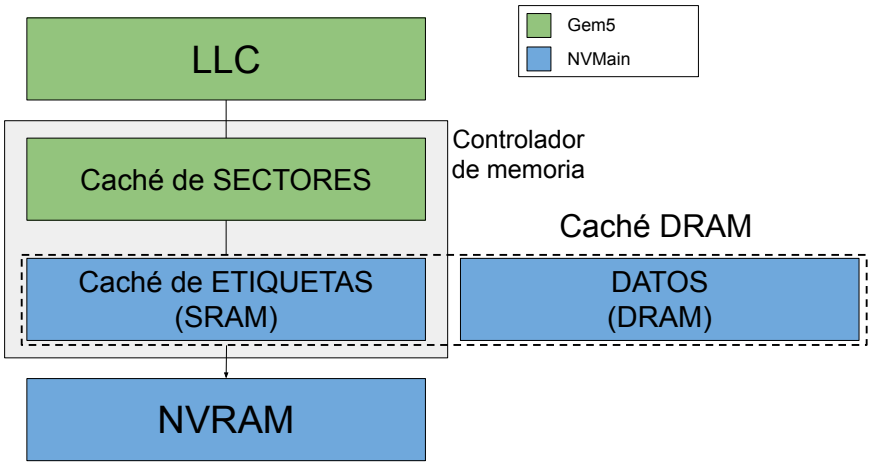

Figura 6.1: Relación entre los componentes y el simulador que los implementa.

6.1 Caché de sectores

En gem5, las cachés se crean utilizando diferentes componentes (*simbojects*) y parámetros. Estos, junto con otros parámetros relativos a la simulación, se pueden personalizar mediante los ficheros de configuración, lo que permite ajustar los diferentes componentes dependiendo del experimento deseado.

Para implementar esta caché se va a definir un nuevo componente con los parámetros necesarios. Para ello, en «gem5/configs/common/Caches.py» y tras la definición de la clase «L2Cache», se define la siguiente clase:

```
1 class HMCSectorCache(Cache):
2     assoc = 16
3     tag_latency = 17
4     data_latency = 17
5     response_latency = 10
6     mshrs = 32
7     tgts_per_mshr = 12
8     write_buffers = 32
9     can_write = False
```

Con esta clase se puede generar un objeto de tipo Cache cuyo grado de asociatividad será 16 y una latencia de etiqueta y datos de 17 ciclos de procesador. También se definen otros parámetros como el tamaño de los *buffers* de escritura («write_buffers») o el número de *miss status hit registers* o simplemente «MSHR», que limita el número máximo de accesos a niveles inferiores de la jerarquía de memoria.

Como se puede observar, faltan parámetros clave por definir, como puede ser el tamaño de la caché. Pese a que no se definen de forma explícita, estos parámetros son heredados de la clase «Cache» que a su vez los hereda de «BaseCache», ambas definidas en «gem5/src/mem/cache/Cache.py». Como es tedioso modificar el parámetro en el fichero de configuración que indica el tamaño de la caché para cada experimento, se definen argumentos en la línea de órdenes del *script* que lanza simulaciones en modo de sistema completo (fichero «fs.py» ubicado en «gem5/configs/example»). Estos argumentos se declaran en «gem5/configs/common/Options.py» y permiten simular diversas configuraciones tan sólo cambiando el valor de los argumentos en la línea de órdenes. Para esto, hay que añadir las siguientes líneas:

```
parser.add_option("--HMCSectorcache", action="store_true")
parser.add_option("--HMCSectorBPS", type="int", default=4)
parser.add_option("--HMCSector_size", type="string", default="8MB")
parser.add_option("--HMCSector_assoc", type="int", default=32)
```

Los valores por defecto definidos en las opciones sobrescribirán los valores heredados de las clases, ya que son los valores con los que se realizará una instancia de la clase «HMCSectorCache». Las opciones definidas son:

- -HMCSectorcache". Denota el uso de la caché de sectores en la simulación.
- -HMCSectorBPS". Indica la cantidad de bloques que forman el sector (un bloque está formado por 64 bytes de datos).
- -HMCSector_size". Define el tamaño total de la caché de sectores.
- -HMCSector_assoc". Fija el grado de asociatividad de la caché de sectores.

Finalmente, solo es necesario añadir la caché a la simulación e interconectarla entre el bus de memoria y la caché de nivel 2 del sistema. Esto se puede hacer en «gem5/configs/common/CacheConfig.py» usando el objeto «options», encargado de interpretar la línea de comandos. Para ello hay que conectar el puerto de salida de la caché de nivel 2 al puerto de entrada de la caché de sectores mediante una red *crossbar*, instanciar la caché de sectores con los parámetros definidos en «Options.py» y finalmente conectar el puerto de salida de la caché de sectores al bus de memoria. Todo esto se puede conseguir en la función «config_cache(...)», cuando se hace referencia al uso de la caché de nivel 2 («if options.l2cache:»), reemplazando esa estructura condicional y todo el contenido del mismo con:

```
if options.l2cache and options.HMCSectorcache:
    system.l2 = l2_cache_class(clk_domain=system.cpu_clk_domain,
                               size=options.l2_size,
                               assoc=options.l2_assoc)
    system.l3 = HMCSectorCache(clk_domain=system.cpu_clk_domain,
                               size=options.HMCSector_size,
                               assoc=options.HMCSector_assoc,
                               tags=SectorTags(num_blocks_per_sector=
                                ↪  options.HMCSectorBPS))

```

```
10      system.tol2bus = L2XBar(clk_domain = system.cpu_clk_domain)
11      system.tol3bus = L2XBar(clk_domain = system.cpu_clk_domain)
12
13      system.l2.cpu_side = system.tol2bus.master
14      system.l2.mem_side = system.tol3bus.slave
15
16      system.l3.cpu_side = system.tol3bus.master
17      system.l3.mem_side = system.membus.slave
18  elif options.l2cache:
19      # Provide a clock for the L2 and the L1-to-L2 bus here as they
20      # are not connected using addTwoLevelCacheHierarchy. Use the
21      # same clock as the CPUs.
22      system.l2 = l2_cache_class(clk_domain=system.cpu_clk_domain,
23                                 **_get_cache_opts('l2', options),
24                                 can_write=options.cache_trace,
25                                 cache_target=options.cache_target)
26
27      system.tol2bus = L2XBar(clk_domain = system.cpu_clk_domain)
28      system.l2.cpu_side = system.tol2bus.master
29      system.l2.mem_side = system.membus.slave
```

6.2 Caché DRAM

A la hora de realizar la implementación de la caché dividida es necesario tener en cuenta que será necesario modificar las direcciones físicas de los accesos a memoria principal para redireccionarlos al canal con tecnología DRAM cuando corresponda. Para ello hay que encontrar un punto en el código del simulador en el que las direcciones no se hayan interpretado todavía. Tal punto se puede encontrar en el bucle principal de NVMain, ubicado en «NVmain/NVM/nvmain.cpp», más concretamente en la función «IssueCommand(...)». En esta ubicación, los accesos generados por gem5 ya han sido traducidos a peticiones de NVMain, pero no se han lanzado al *pipeline* de la memoria principal, siendo susceptibles de modificaciones.

6.2.1. Implementación en tres fases

El espectro de posibilidades de diseño de la caché dividida es muy amplio y plantea muchas cuestiones. Por ejemplo ¿La relación entre la caché de sectores y la caché DRAM debería ser de inclusividad o exclusividad? ¿Qué sucede con los bloques a escribir que se reciben de gem5? ¿Cuándo y cómo se deben reemplazar los bloques de esta caché? Estas cuestiones reflejan parte de la complejidad del sistema. En este contexto, con tal de facilitar de implementación, se define un plan en tres fases que permite abordarla de forma incremental, añadiendo interacciones cada vez más complejas entre los simuladores y los componentes. Las fases son las siguientes:

1. NVRAM *Fetch*. Inicialmente, todos los datos se almacenan directamente en memoria principal, es decir, en NVRAM. En esta fase, cuando el procesador intenta acceder a esos datos, no se encuentran almacenados en ningún nivel de la jerarquía de caché, realizando una petición a NVRAM que acaba devolviendo los datos necesarios.

2. *Writeback*. Al avanzar la ejecución, cada vez más datos se llevan a la caché de sectores, hasta que se llena y comienzan a darse reemplazos, sectores que una política de reemplazo marca como menos útiles y que por lo tanto pueden reemplazarse por nuevo sectores. Los sectores reemplazados y modificados se almacenan en la caché dividida que actúa como caché víctima. Además, puede darse el caso de que la caché víctima acabe llenándose, lo que implica más reemplazos y escrituras en la memoria principal (NVRAM).

3. DRAM *Fetch*. En caso de acierto en la caché de etiquetas, se accede a los datos desde DRAM y no desde NVRAM.

A continuación se detalla cada una de las fases, así como los cambios realizados en el código para soportarlos.

NVRAM *Fetch*

Tras recibir un acceso de lectura desde la caché de sectores, la memoria NVRAM sirve la petición. Para conseguir esto, se realizan dos pasos:

1. Reencaminar el acceso. Debido al funcionamiento de la caché dividida, es necesario control total sobre el contenido del canal con memoria DRAM. Por ello, el primer

paso es encaminar a NVRAM todos los accesos originalmente destinados a direcciones DRAM, dejando así la memoria DRAM disponible para los propósitos de la caché dividida.

2. Servir los accesos. NVRAM sirve **todos** los accesos emitidos desde la caché de sectores.

Writeback

Tras servir los accesos de lectura mediante NVRAM, el siguiente paso es servir los *writebacks* correspondientes a los sectores víctima de la caché de sectores. Para implementar la lógica que se encargue de esto, se han definido los siguientes pasos:

1. Revisar si el bloque accedido ya está presente en la caché dividida. Es decir, es necesario conocer la ubicación del bloque (canal de memoria DRAM o canal de memoria NVRAM).
2. Escritura DRAM. En caso de que el bloque esté presente en DRAM, se modifica el acceso para que se dirija al canal de memoria DRAM. En otro caso, es necesario reservar una entrada de la caché dividida o realizar un *writeback* de DRAM a NVRAM por falta de espacio en esta caché. En este caso, los datos reemplazados se leen de DRAM, y tras ello se escriben en NVRAM, para finalmente realizar el *writeback* de la caché de sectores en la caché DRAM.
3. Actualizar la información en las etiquetas de la caché dividida. Cuando se realiza un *writeback* de DRAM a NVRAM, se actualiza la información de la caché de etiquetas para indicar la nueva ubicación de los datos.

DRAM *Fetch*

Una vez la DRAM almacena datos provenientes de la caché de sectores, esta puede comenzar a servir los accesos que antes servía la NVRAM. De esta forma no solo se evita el acceso a NVRAM sino que se accede a una memoria alrededor de 10 veces más rápida. Para ello se han ideado tres pasos:

1. Revisar las etiquetas de la caché dividida para comprobar la disponibilidad en DRAM. De forma similar que al realizar *writeback* desde la caché de sectores, es necesario revisar si el bloque ya se encuentra presente en DRAM para redireccionar los accesos (tanto lecturas como escrituras) al canal correcto.

2. Emitir el acceso a la memoria correspondiente. Si el bloque se encuentra disponible (acierto), el acceso se modifica para que acceda a la dirección apropiada en DRAM. En caso contrario (fallo), se lanza la petición de lectura a NVRAM o bien se ejecuta la fase de *writeback* explicada anteriormente.

3. Finalmente, la información de las etiquetas de la caché dividida se actualiza.

Trabajo adicional

Además de la implementación descrita en las tres fases anteriores, se han implementado dos modificaciones adicionales:

1. Almacenar una copia de los bloques leídos desde NVRAM en la caché DRAM. Es probable que sea necesario volver a acceder al mismo bloque de memoria, y por lo tanto almacenarlo en la memoria más rápida puede ofrecer mayores beneficios a lo largo de la ejecución.

2. Se añade una latencia adicional de 4 ns (ocho ciclos de procesador a 2 GHz) a los accesos a la caché de etiquetas. De esta forma se simula la latencia de acceder a una memoria SRAM para consultar las etiquetas almacenadas en la caché dividida.

6.2.2. Código desarrollado

En la función «IsIssuable(...)» «NVmain/NVM/nvmain.cpp» tras el bloque «if(Check Prefetch(request))» se implementa la lógica encargada de redireccionar los accesos o bien a DRAM o a NVRAM. Para ello se consulta si la etiqueta está presente en la caché de etiquetas, así como la dirección que ocupa en el dispositivo DRAM si está presente (línea 8).

En caso de que no se encuentre en DRAM, entonces (línea 10):

- Las accesos de escritura deben actualizar el contenido de la DRAM. Existen dos casos diferenciados, que no exista espacio en la caché (línea 16), en cuyo caso habrá que realizar un reemplazo para lo cual es necesario realizar una lectura de los datos a reemplazar, para posteriormente escribirlos en NVRAM. Esta tarea se realiza preparando un acceso de escritura de antemano (línea 28 a 38), que se ejecutará una vez la lectura se haya completado; y preparando la petición de lectura a DRAM (línea 46 a 57) para ejeecutarla posteriormente (línea 60). Si por el contrario, existe

espacio en la caché DRAM, simplemente se busca una ubicación no ocupada (línea 72 a 92) y se realiza la escritura a esa ubicación.

- Los accesos de lectura se ejecutan sin modificar (línea 95 a 100)

En caso de que la etiqueta del bloque se encuentre presente en la caché de etiquetas, se modifica la dirección del acceso para que se ejecute sobre la dirección almacenada (línea 103 a 120). El código referenciado se puede observar en el anexo D.

En la función «RequestComplete(...)» «NVmain/NVM/nvmain.cpp» tras la línea «bool rv = false;» continúa la implementación de «IsIssuable(...)», lanzando de forma paralela la escritura en NVRAM del bloque de datos leído de DRAM para realizar el reemplazo (línea 181 a 201) y la escritura de los datos nuevos al bloque recién liberado en DRAM (línea 168 a 176), ya que esta petición se interceptó y no se permitió que se ejecutase. Además, los datos de las lecturas de NVRAM se copian y se añaden a la DRAM, debido a que es probable que se vuelvan a utilizar en un futuro (línea 1 a 139), como se trata de modificar la caché DRAM, el código es muy similar al explicado en «IsIssuable()» y por ello no se entrará en más detalles. Todo el código discutido está disponible en el anexo E.

En «NVmain/NVM/nvmain.h» tras la línea «void GeneratePrefetches(NVMainRequest *request, std::vector<NVMAddress>& prefetchList);» se incorporan variables y diversas estructuras de datos con tal de guardar estadísticas, mantener listas de peticiones en espera o configurar el tamaño de la caché (línea 1 a 21). Además se incorporar la clase «LRU» (línea 23 a 100), encargada de implementar el algoritmo LRU, utilizado como algoritmo de reemplazo en la caché, parte importante de la lógica de la caché de etiquetas. En el anexo F se puede encontrar el código fuente desarrollado para el fichero «NVmain/NVM/nvmain.h».

Cambios menores

Además de los cambios mencionados en la sección anterior, se han incluido pequeños cambios a la hora de interpretar los ficheros de configuración con tal de poder elegir si la caché DRAM debía entrar en funcionamiento, así como cambios en otras clases para facilitar la implementación como los *flags* «NVMainRequest::FLAG_EVICT» y «NVMainRequest::FLAG_SKIP_TAG».

CAPÍTULO 7

Resultados experimentales

En este capítulo se evalúa el rendimiento de la propuesta explicada en el capítulo 5, así como el efecto que cada uno de los componentes implementados tiene sobre las prestaciones del sistema global.

7.1 Sistemas simulados

Para realizar la evaluación, se efectúan experimentos con varios sistemas, cada uno con una configuración diferente en el sistema de memoria. Cada configuración introduce una modificación incremental sobre la anterior, lo que permite observar los beneficios aportados por cada modificación de forma independiente. Finalmente, se experimenta con la propuesta totalmente implementada.

Las características principales de cada uno de los sistemas simulados por gem5 se pueden observar en la tabla 7.1. Estas características son comunes a todos los sistemas, ya que los cambios solo se realizan a nivel arquitectural y no temporal del subsistema de memoria.

La configuración del subsistema de memoria variará según la tabla 7.2. En esta se definen en más detalle la configuración de los cuatro sistemas simulados («DRAM», «NV», «NV-S», y «NV-S-D»). En general, todos los sistemas disponen de una caché de nivel uno tanto de instrucciones como de datos, así como una caché de nivel dos. El resto de cachés se van añadiendo incrementalmente a cada uno de los sistemas, desde el sistema que no incluye ninguna caché adicional («DRAM») hasta el que las incluye todas («NV-S-D»). Además, la memoria principal es diferente según el sistema a simular. Como pretendemos mejorar las prestaciones de un sistema con memoria principal no volátil, los sistemas que

Componente	Parámetro	Valor
CPU	Tipo	TimingSimpleCPU
	Núcleos	1
	Frecuencia	2.0 GHz
Caché L1 Datos	Tamaño, tamaño de bloque, Asociatividad	32 KB, 64 B, 8-vias
	Latencia etiqueta/datos	3/3 ciclos
	MSHRs/TGTS por MSHR	4/20
	Nº de puertos de lectura / Nº de puertos de escritura	1/1
	Política de escritura	*Write-back*
	Política de reemplazo	LRU
Caché L1 Instr.	Tamaño, tamaño de bloque, Asociatividad	32 KB, 64 B, 8-vias
	Latencia etiqueta/datos	3/3 ciclos
	MSHRs/TGTS per MSHR	4/20
	Nº de puertos de lectura / Nº de puertos de escritura	1/1
	Exclusivamente de lectura	Sí
	Writeback clean	Sí
	Política de escrituras	*Write-back*
	Política de reemplazo	LRU
Caché L2	Tamaño, tamaño de bloque, Asociatividad	2 MB, 64 B, 16-vias
	Latencia etiqueta/datos	11/11 ciclos
	MSHRs/TGTS per MSHR	20/12
	Write buffers	8
	Nº de puertos de lectura / Nº de puertos de escritura	2/2
	Política de escrituras	*Write-back*
	Política de reemplazo	LRU
Caché de sectores	Tamaño, tamaño de bloque, Asociatividad	8 MB, 256 B, 16-vias
	Latencia etiqueta/datos	17/17 ciclos
	MSHRs/TGTS per MSHR	32/12
	Write buffers	32
	Nº de puertos de lectura / Nº de puertos de escritura	2/2
	Política de escrituras	*Write-back*
	Política de reemplazo	LRU
Caché dividida	Canales de memoria	2
	Capacidad de almacenamiento	64MB
	Política de reemplazo	LRU

Tabla 7.1: Parámetros de configuración del sistema de gem5 para una tecnología de nodo de 22 nm. Los ciclos han sido calculados para una frecuencia de reloj de 2 GHz.

dispongan de caché de sectores incluirán tecnología NVRAM. Si además integran caché dividida, también implementarán DRAM para almacenar los datos de esta caché.

Caché\Sistema	DRAM	NV	NV-S	NV-S-D
Caché L1D	S	S	S	S
Caché L1I	S	S	S	S
Caché L2	S	S	S	S
Caché de sectores	N	N	S	S
Caché dividida	N	N	N	S
Tipo de dispositivo de memoria	DRAM	NVRAM		NVRAM / DRAM
Nº de canales de memoria	2			
Capacidad por canal (GB)	4	32		32 / 4
Capacidad total (GB)	8	64		36
Frecuencia de memoria (MHz)	1466	488		488 / 3800

Tabla 7.2: Configuración de la jerarquía de memoria para los sistemas estudiados. «S» denota la presencia de la estructura y «N» la ausencia.

Con respecto al número de canales de memoria principal, el sistema «NV» consta de dos canales de memoria no volátil mientras que el sistema «DRAM» implementa dos canales de memoria DRAM DDR4. Los sistemas «NV-S» y «NV-S-D» son variaciones del sistema «NV» con los cambios planteados en el capítulo 6. «NV-S» incluye una caché de sectores y «NV-S-D» además añade una caché dividida para gestionar una organización de memoria híbrida NVRAM-DRAM.

En cuanto a los resultados, se obtendrán diferentes estadísticas de las simulaciones como ciclos, instrucciones, accesos a memoria, etc. Con estas estadísticas se obtienen medidas de prestaciones como las instrucciones por ciclo o IPC. Usando estas medidas se realizará la comparación entre los diferentes sistemas para determinar la mejora o el deterioro de los cambios sobre el sistema.

7.2 Análisis de resultados

En esta sección se presentan y se analizan los resultados de las diferentes simulaciones realizadas en los cuatro sistemas presentados en la sección anterior. Para este análisis, el sistema «NV» será la referencia en cuanto a prestaciones. Se espera que las prestaciones de todos los sistemas que utilizan memoria no volátil sean peores que las del sistema que

utiliza tecnología DRAM, ya que la memoria NVRAM modelada es en torno a tres veces más lenta.

Primero, se presentan los resultados para un sistema convencional que utiliza DRAM. La tabla 7.3 presenta sus resultados. Se presentan estadísticas de rendimiento general del sistema así como del subsistema de memoria, ya que no contiene ningún otro componente de los desarrollados. La cantidad de lecturas es entre dos y tres veces mayor que la de escrituras. Además, la latencia media de DRAM es ligeramente superior a 100 ciclos de procesador, dentro del rango normal para un procesador de 2 GHz. Se puede observar como «MEMTIER» tiene una latencia un 15,6 % mayor que «REDIS», lo que apunta a que «MEMTIER» estresa algo más la memoria.

	Estadística (Unidades)	REDIS	MYSQL	MEMTIER
Rendimiento General	Nº Instrucciones	6384602960	9076068212	16961006304
	Nº Ciclos	4,56E+10	6,03E+10	1,22E+11
	IPC	0,14	0,15	0,14
DIMM DRAM	Lecturas DRAM (Bloques)	177655	5248072	93023276
	Escrituras DRAM (Bloques)	123106	2038872	44474054
	Latencia Media DRAM (Ciclos NVMain)	76	81	89
	Latencia Media DRAM (Ciclos CPU)	103,68	110,50	121,42

Tabla 7.3: Estadísticas de simulación para el sistema «DRAM».
Este sistema solo contiene DIMM DRAM.

	Estadística (Unidades)	REDIS	MYSQL	MEMTIER
Rendimiento General	Nº Instrucciones	6371915577	8944449586	14711377877
	Nº Ciclos	4,56E+10	6,11E+10	1,22E+11
	IPC	0,14	0,14	0,12
DIMM NVRAM	Lecturas NVRAM (Bloques)	215844	8152549	71632158
	Escrituras NVRAM (Bloques)	132912	3223444	32692182
	Latencia Media NVRAM (Ciclos NVMain)	88	91	107
	Latencia Media NVRAM (Ciclos CPU)	360,66	372,95	438,52

Tabla 7.4: Estadísticas de simulación para el sistema NV.
Este sistema solo contiene DIMM NVRAM.

Respecto al sistema «NV», la tabla 7.4 contiene las mismas estadísticas que el sistema DRAM. Comparando ambos, se puede observar como la latencia de memoria del sistema con tecnología NVRAM es entorno a tres veces mayor que la de DRAM. A pesar de la

gran diferencia en latencia, el IPC apenas varia, con la única expepción de «MEMTIER», que es el *benchmark* que más estresa la memoria.

La tabla 7.5 muestra los resultados obtenidos para el sistema «NV-S». Esta tabla muestra cómo de eficiente es la caché de sectores para cada aplicación. Las tres aplicaciones muestran una tasa de aciertos muy diferente, que varía del 71,18 % en «REDIS» al 37,97 % en «MEMTIER». Sin embargo, estos valores no son suficientes para ocultar la latencia de memoria de los dispositivos NVRAM y mejorar el rendimiento general del sistema de forma significativa.

	Estadística (Unidades)	REDIS	MYSQL	MEMTIER
Rendimiento General	Nº Instrucciones	6386117677	9121590348	15648837411
	Nº Ciclos	4,59E+10	6,04E+10	1,22E+11
	IPC	0.14	0.15	0.13
Caché de sectores	Aciertos	130301	3463789	31117775
	Fallos	52746	1755828	51062939
	Tasa de aciertos	71,18	66,36	37,87
	MPKI	0,006	0,130	1,855
	Reemplazos Silenciosos/Limpios	38275	1248895	33598155
	Sectores reemplazados con un bloque sucio	1831	61184	5337427
	Sectores reemplazados con dos bloques sucios	1370	32653	2385899
	Sectores reemplazados con tres bloques sucios	1120	21286	878267
	Sectores reemplazados con todos los bloques sucios	10318	390668	9802362
	Bloques sucios reemplazados	49203	1753020	51953474
DIMM NVRAM	Lecturas NVRAM (Bloques)	78817	1921721	51066278
	Escrituras NVRAM (Bloques)	31511	1204467	23330333
	Latencia Media NVRAM (Ciclos NVMain)	88	94	108
	Latencia Media NVRAM (Ciclos CPU)	360,66	385,25	442,62

Tabla 7.5: Estadísticas de simulación para el sistema NV-S.
Este sistema no contiene la caché dividida con datos almacenados en DRAM.

Sin embargo es interesante observar la cantidad de peticiones, tanto de lecturas como de escrituras, se reduce entre 1,4 y 3 veces. Esto indica que la caché de sectores puede ayudar a extender la vida útil de los DIMM NVRAM.

También se han incluido los fallos por millar de instrucciones - abreviado como *MPKI* de sus siglas en inglés (*Misses per KiloInstruction*) como parte de las estadísticas de la caché de sectores. El motivo principal es que la tasa de aciertos no cuantifica la presión que ejerce la aplicación en nivel inferior de la jerarquía. A mayor MPKI, mayor ancho de banda consumido en niveles inferiores, lo que puede resultar en un rendimiento peor. Se puede observar como para «NV-S» y «NV-S-D», «MEMTIER» es la aplicación con mayor

MPKI y que muestra la mayor latencia, ya que ejerce mayor presión sobre la memoria principal.

La tabla 7.6 presenta los resultados del sistema «NV-S-D». Estos resultados muestran cómo de eficaz es la propuesta completa para cada aplicación. Al contrario de la caché de sectores, la caché dividida con datos en DRAM tiene un impacto muy positivo en las prestaciones. La latencia media del canal de memoria NVRAM se reduce de forma considerable, lo que apunta a que la jerarquía parece funcionar de forma adecuada en lo que respecta a la ocultación de latencia. Como consecuencia, la propuesta otorga grandes beneficios a las prestaciones.

	Estadística (Unidades)	REDIS	MYSQL	MEMTIER
Rendimiento General	Nº Instrucciones	6427577697	9173715204	17597003715
	Nº Ciclos	4.64E+10	6.03E+10	1.20E+11
	IPC	0.14	0.15	0.15
Caché de Sectores	Aciertos	130412	3534128	42440220
	Fallos	54507	1663828	60721318
	Tasa de aciertos	70.52	67.99	41.14
	MPKI	0.006	0.123	2.060
	Reemplazos Silenciosos/Limpios	39763	1183021	40117102
	Sectores reemplazados con un bloque sucio	1884	59451	6032116
	Sectores reemplazados con dos bloques sucios	1399	32959	2719364
	Sectores reemplazados con tres bloques sucios	1133	21350	998094
	Sectores reemplazados con todos los bloques sucios	10514	368941	11777836
	Bloques sucios reemplazados	50137	1665183	61576470
Caché dividida	Aciertos de Etiqueta	46549	2171234	68876567
	Fallos de Etiqueta	70479	805522	20574889
	Tasa de aciertos de Etiquetas	39.78	72.94	77.00
	Lecturas DRAM (Bloques)	84959	1840785	61107895
	Escrituras DRAM (Bloques)	102548	1941493	48918450
	Latencia Media DRAM (Ciclos NVMain)	101	102	104
	Latencia Media DRAM (Ciclos CPU)	53.16	53.68	54.74
DIMM NVRAM	Lecturas NVRAM (Bloques)	70223	800119	20193669
	Escrituras NVRAM (Bloques)	70479	805522	20574889
	Latencia Media NVRAM (Ciclos NVMain)	65	65	76
	Latencia Media NVRAM (Ciclos CPU)	266.39	266.39	311.48

Tabla 7.6: Estadísticas de simulación para el sistema NV-S-D.

Estos beneficios en prestaciones se pueden observar en la figura 7.1, que presenta el rendimiento general (IPC) de los sistemas «DRAM», «NV-S» y «NV-S-D» respecto al sistema «NV». Es decir, un valor superior a 1 representa la aceleración obtenida (por ejemplo,

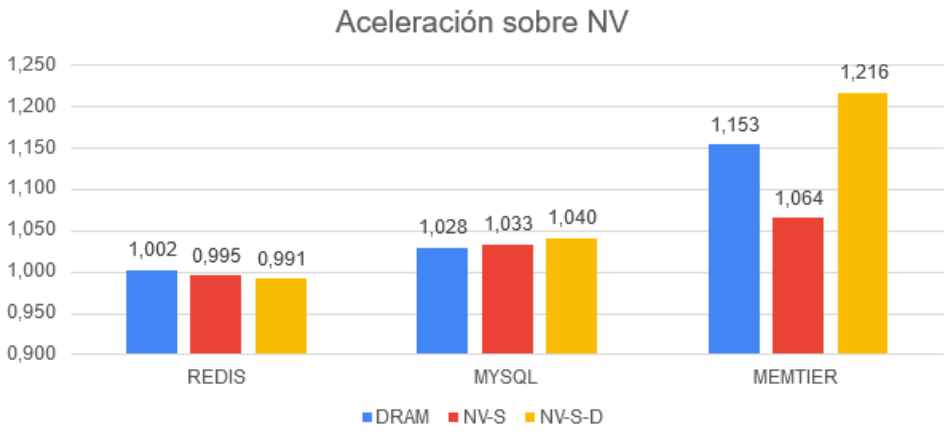

Figura 7.1: Aceleración sobre el sistema NV.

1.1 indica un 10 % de mejora) y un valor inferior a uno significa una ralentización con respecto al sistema «NV».

De esta figura se pueden extraer dos observaciones. En primer lugar, utilizar un único nivel de caché apenas ayuda a ocultar la contención debido a las grandes latencias de NVRAM. Es decir, como las latencias del sistema «NV-S» son similares a las de «NV» las prestaciones globales son similares. Esto efecto se observa especialmente en aplicaciones que no estresan la memoria como «REDIS» y «MYSQL», donde «NV-S» apenas mejora las prestaciones. En segundo lugar, las diferencias significativas en prestaciones se dan en «MEMTIER». Para esta aplicación, un único nivel de caché de sectores no es suficiente lo que indica que una implementación con los dos niveles es necesaria cuando la memoria se encuentra bajo mucho estrés.

CAPÍTULO 8

Conclusiones

A continuación comentamos las principales conclusiones extraídas de este trabajo. Para ello, primero, comprobamos si hemos cumplido los objetivos propuestos en la sección 1.3; segundo, realizaremos una revisión general del trabajo realizado; y, finalmente, concluiremos con posibles ampliaciones del trabajo a realizar en el futuro.

8.1 Objetivos alcanzados

En esta sección se expondrán de nuevo los objetivos que planteamos en la sección 1.3 y, en base a estos, expondremos si se han llevado a cabo o no de forma satisfactoria junto con una justificación.

1. **Diseñar una arquitectura de memoria para ocultar la latencia de los módulos NVRAM.** Este objetivo se ha llevado a cabo de forma satisfactoria ya que hemos definido la arquitectura de memoria ideada así como los componentes necesarios, y la función de cada uno de ellos. Además, hemos discutido la relación entre ellos (ver capítulo 5).

2. **Implementar la arquitectura de memoria diseñada en el primer objetivo.** En el capítulo 6 se detallan los pasos y modificaciones realizadas en los simuladores con tal de implementar la arquitectura de memoria propuesta. Por lo tanto, pensamos que este objetivo se ha cumplido exitosamente.

3. **Realizar un análisis de las prestaciones del sistema con la nueva arquitectura.** En la sección 7.2 se presentan los resultados del estudio realizado sobre el impacto de la arquitectura de memoria en las prestaciones del sistema, comparándolo con varias

arquitecturas diferentes y analizando el efecto de cada uno de los componentes. Así pues, se ha cumplido este objetivo.

8.2 Visión general del trabajo realizado

En este TFM se han mostrado los principales componentes de un controlador de memoria híbrido que utiliza NVRAM como memoria principal. Esta tecnología de memoria soporta una capacidad de almacenamiento mucho mayor, que se alinea mejor con los requisitos crecientes de muchas aplicaciones modernas, especialmente en el ámbito de la computación en la nube e inteligencia artificial. Sin embargo, el tiempo de acceso de estas memorias es mucho mayor que el de las memorias DRAM convencionales. En consecuencia, son necesarias soluciones arquitecturales para ocultar la latencia de los accesos a memoria principal con tal de mejorar las prestaciones del subsistema de memoria.

Se ha propuesto una primera versión del diseño, que consiste en una arquitectura de dos niveles de caché en el subsistema de memoria principal. El primer nivel consiste en una caché SRAM organizada en sectores (conjuntos de bloques de 64 B) y el segundo nivel está compuesto por una caché DRAM. Asimismo, hemos definido las interacciones entre estos componentes y los módulos de memoria principal. Como resultado de esto, un fallo en el primer nivel puede resultar en varios accesos a memoria para mantener la coherencia.

Finalmente, se han extraído algunas ideas clave como resultado de esta investigación:

1. Un fallo en la caché de sectores puede disparar varios accesos a la caché DRAM. Por lo tanto la caché de sectores debe ser significativamente más rápida que DRAM disponible comercialmente para mantener el rendimiento.

2. De forma similar a la caché de sectores, la memoria DRAM utilizada para implementar los datos de la caché DRAM debe ser más rápida que la DRAM disponible comercialmente ya que, además de necesitarse varios accesos por un único fallo de la caché de sectores, las etiquetas y los datos de la caché DRAM se acceden de forma secuencial.

3. El diseño de dos niveles es necesario para aplicaciones con una gran demanda de memoria principal (como «MEMTIER»).

4. La mayor latencia es aquella introducida por la memoria NVRAM, que incrementa la latencia del subsistema de memoria de forma significativa al aumentar la contención en aplicaciones que generan mucha presión sobre la memoria principal.

8.3 Trabajo futuro

Si bien se ha conseguido un aumento sensible en las prestaciones del subsistema de memoria gracias a las modificaciones planteadas en este trabajo todavía quedan diversas líneas de investigación que pueden resultar interesantes. Hemos localizado tres principales vías de exploración:

- Incrementar el estrés sobre la memoria. Existen diversas formas de realizar esto, por ejemplo, modificando los parámetros de las aplicaciones utilizadas como *benchmarks* o modelando procesadores multinúcleo o fuera de orden.
- Probar a incrementar el número de canales y diferentes políticas de traducción de direcciones. Las pruebas realizadas han sido llevadas a cabo con dos canales en cada sistema pero cabe estudiar la escalabilidad de la propuesta más canales de memoria.
- Estudiar el impacto de cachés con tecnología embedded DRAM (eDRAM). Las memorias eDRAM se integran sobre chips con tecnología CMOS como podría ser el propio procesador. De esta forma las prestaciones mejoran significativamente ya que la latencia es sustancialmente menor. Además, este estudio se puede realizar sobre otro tipo de memorias que permita modelar NVMain. Por ejemplo otras versiones de memoria no volátil como memorias de acceso aleatorio magnetoresistivas (también conocidas como MRAM de sus siglas en inglés).
- Analizar los diferentes componentes que conforman la latencia de acceso a la memoria para de identificar posibles mejoras. De esta forma se profundizaría en más campos de estudio del diseño del controlador con el objetivo de aumentar todavía más las prestaciones que este ofrece.

Bibliografía

[1] Simon Segars. Arm processor evolution: Bringing high performance to mobile devices. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–37, 2011.

[2] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 152–165, 2017.

[3] Niladrish Chatterjee, Manjunath Shevgoor, Rajeev Balasubramonian, Al Davis, Zhen Fang, Ramesh Illikkal, and Ravi Iyer. Leveraging heterogeneity in dram main memories to accelerate critical word access. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24, 2012.

[4] Chia Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2014.

[5] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[6] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 467–478, 2014.

[7] Sujay Phadke and Satish Narayanasamy. Mlp aware heterogeneous memory system. In *2011 Design, Automation Test in Europe*, pages 1–6, 2011.

[8] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient dram caching via software/hardware cooperation. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2017.

[9] Miguel Antonio Avargues Gutiérrez. Análisis de requerimientos y diseño de un controlador de memoria principal no volátil.

[10] 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.

[11] 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.

[12] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[14] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, 2012.

[15] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 404–415, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37, 2014.

[17] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. A fully associative, tagless dram cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 211–222, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Cheng-Chieh Huang and Vijay Nagarajan. Atcache: Reducing dram cache latency via a small sram tag cache. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 51–60, New York, NY, USA, 2014. Association for Computing Machinery.

[19] 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.

[20] 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.

[21] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

APÉNDICE A

Código fuente del fichero «nvmain_mem.hh»

```
/*
 * Copyright (c) 2012-2014 Pennsylvania State University
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met: redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer;
 * redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution;
 * neither the name of the copyright holders nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
```

```
22  * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
23  * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
24  * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
25  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
26  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
27  */
28
29 /*
30  *  This file is part of NVMain- A cycle accurate timing, bit-accurate
31  *  energy simulator for non-volatile memory. Originally developed by
32  *  Matt Poremba at the Pennsylvania State University.
33  *
34  *  Website: http://www.cse.psu.edu/~poremba/nvmain/
35  *  Email: mrp5060@psu.edu
36  *
37  *  ---------------------------------------------------------------------
38  *
39  *  If you use this software for publishable research, please include
40  *  the original NVMain paper in the citation list and mention the use
41  *  of NVMain.
42  *
43  */
44
45 #include "SimInterface/Gem5Interface/Gem5Interface.h"
46 #include "Simulators/gem5/nvmain_mem.hh"
47 #include "Utils/HookFactory.h"
48
49 #include "base/random.hh"
50 #include "base/statistics.hh"
51 #include "debug/NVMain.hh"
52 #include "debug/NVMainMin.hh"
53 #include "config/the_isa.hh"
54
55 using namespace NVM;
56
```

```
57 // This members are singleton values used to hold the main instance of
58 // NVMain and it's wake/sleep (i.e., timing/atomic) status. These are
59 // needed since NVMain assumes a contiguous address range while gem5
60 // ISAs generally do not. The multiple instances allow for the gem5
61 // AddrRanges to be used normally while this class remapped to NVMains
62 // contiguous region.
63 NVMainMemory *NVMainMemory::masterInstance = NULL;
64
65 NVMainMemory::NVMainMemory(const Params &p)
66     : AbstractMemory(p), clockEvent(this), respondEvent(this),
67       drainManager(NULL), lat(p.atomic_latency),
68       lat_var(p.atomic_variance), nvmain_atomic(p.atomic_mode),
69       NVMainWarmUp(p.NVMainWarmUp), port(name() + ".port", *this)
70 {
71     char *cfgparams;
72     char *cfgvalues;
73     char *cparam, *cvalue;
74
75     char *saveptr1, *saveptr2;
76
77     nextEventCycle = 0;
78
79     m_nvmainPtr = NULL;
80     m_nacked_requests = false;
81
82     m_nvmainConfigPath = p.config;
83
84     m_nvmainConfig = new Config( );
85
86     m_nvmainConfig->Read( m_nvmainConfigPath );
87     std::cout << "NVMainControl: Reading NVMain config file: " <<
       ↪  m_nvmainConfigPath << "." << std::endl;
88
89     clock = clockPeriod( );
90
```

```
91      m_avgAtomicLatency = 100.0f;
92      m_numAtomicAccesses = 0;
93
94      retryRead = false;
95      retryWrite = false;
96      retryResp = false;
97      sync = false;
98      retryflag = false;
99      m_requests_outstanding = 0;
100
101     /*
102      * Modified by Tao @ 01/22/2013
103      * multiple parameters can be manually specified
104      * please separate the parameters by comma ","
105      * For example,
106      *    configparams = tRCD,tCAS,tRP
107      *    configvalues = 8,8,8
108      */
109     cfgparams = (char *)p.configparams.c_str();
110     cfgvalues = (char *)p.configvalues.c_str();
111
112     for( cparam = strtok_r( cfgparams, ",", &saveptr1 ), cvalue = strtok_r(
        ↪  cfgvalues, ",", &saveptr2 )
113             ; (cparam && cvalue) ; cparam = strtok_r( NULL, ",", &saveptr1
                ↪  ), cvalue = strtok_r( NULL, ",", &saveptr2) )
114     {
115         std::cout << "NVMain: Overriding parameter `" << cparam << "' with
            ↪  `" << cvalue << "'" << std::endl;
116         m_nvmainConfig->SetValue( cparam, cvalue );
117     }
118
119    BusWidth = m_nvmainConfig->GetValue( "BusWidth" );
120    tBURST = m_nvmainConfig->GetValue( "tBURST" );
121    RATE = m_nvmainConfig->GetValue( "RATE" );
122
```

```
123     lastWakeup = curTick();
124     startWakeup = curTick();
125 }
126
127
128 NVMainMemory::~NVMainMemory()
129 {
130     std::cout << "NVMain dtor called" << std::endl;
131 }
132
133
134 void
135 NVMainMemory::init()
136 {
137     if (!port.isConnected()) {
138         fatal("NVMainMemory %s is unconnected!\n", name());
139     } else {
140         port.sendRangeChange();
141     }
142
143     if( masterInstance == NULL )
144     {
145         masterInstance = this;
146
147         m_nvmainPtr = new NVM::NVMain( );
148         m_statsPtr = new NVM::Stats( );
149         m_nvmainSimInterface = new NVM::Gem5Interface( );
150         m_nvmainEventQueue = new NVM::EventQueue( );
151         m_nvmainGlobalEventQueue = new NVM::GlobalEventQueue( );
152         m_tagGenerator = new NVM::TagGenerator( 1000 );
153
154         m_nvmainConfig->SetSimInterface( m_nvmainSimInterface );
155
156         statPrinter.nvmainPtr = m_nvmainPtr;
157         statReseter.nvmainPtr = m_nvmainPtr;
```

```
158
159         if( m_nvmainConfig->KeyExists( "StatsFile" ) )
160         {
161             statPrinter.statStream.open( m_nvmainConfig->GetString(
                ↪  "StatsFile" ).c_str(),
162                                          std::ofstream::out |
                                             ↪  std::ofstream::app );
163         }
164
165         statPrinter.memory = this;
166         statPrinter.forgdb = this;
167
168         //registerExitCallback( &statPrinter );
169         ::Stats::registerDumpCallback([this]() { statPrinter.process(); });
170         ::Stats::registerResetCallback([this]() { statReseter.process();
            ↪  });
171
172         SetEventQueue( m_nvmainEventQueue );
173         SetStats( m_statsPtr );
174         SetTagGenerator( m_tagGenerator );
175
176         m_nvmainGlobalEventQueue->SetFrequency( m_nvmainConfig->GetEnergy(
            ↪  "CPUFreq" ) * 1000000.0 );
177         SetGlobalEventQueue( m_nvmainGlobalEventQueue );
178
179         clock = static_cast<Tick>(round(1000000 /
            ↪  m_nvmainConfig->GetEnergy("CPUFreq")));
180         // TODO: Confirm global event queue frequency is the same as this
            ↪  SimObject's clock.
181
182         /*  Add any specified hooks */
183         std::vector<std::string>& hookList = m_nvmainConfig->GetHooks( );
184
185         for( size_t i = 0; i < hookList.size( ); i++ )
186         {
```

```
187             std::cout << "Creating hook " << hookList[i] << std::endl;
188
189             NVMObject *hook = HookFactory::CreateHook( hookList[i] );
190
191             if( hook != NULL )
192             {
193                 AddHook( hook );
194                 hook->SetParent( this );
195                 hook->Init( m_nvmainConfig );
196             }
197             else
198             {
199                 std::cout << "Warning: Could not create a hook named `"
200                     << hookList[i] << "'." << std::endl;
201             }
202         }
203
204         /* Setup child and parent modules. */
205         AddChild( m_nvmainPtr );
206         m_nvmainPtr->SetParent( this );
207         m_nvmainGlobalEventQueue->AddSystem( m_nvmainPtr, m_nvmainConfig );
208         m_nvmainPtr->SetConfig( m_nvmainConfig );
209
210         masterInstance->allInstances.push_back(this);
211     }
212     else
213     {
214         masterInstance->allInstances.push_back(this);
215         masterInstance->otherInstance = this;
216     }
217 }
218
219
220 void NVMainMemory::startup()
221 {
```

```
222     DPRINTF(NVMain, "NVMainMemory: startup() called.\n");
223     DPRINTF(NVMainMin, "NVMainMemory: startup() called.\n");
224 
225     /*
226      *  Schedule the initial event. Needed for warmup and timing mode.
227      *  If we are in atomic/fast-forward, wakeup will be disabled upon
228      *  the first atomic request receieved in recvAtomic().
229      */
230     if (!masterInstance->clockEvent.scheduled())
231         schedule(masterInstance->clockEvent, curTick() + clock);
232 
233     lastWakeup = curTick();
234     startWakeup = curTick();
235 }
236 
237 
238 void NVMainMemory::wakeup()
239 {
240     DPRINTF(NVMain, "NVMainMemory: wakeup() called.\n");
241     DPRINTF(NVMainMin, "NVMainMemory: wakeup() called.\n");
242 
243     schedule(masterInstance->clockEvent, clockEdge());
244 
245     lastWakeup = curTick();
246 }
247 
248 
249 Port &
250 NVMainMemory::getPort(const std::string& if_name, PortID idx)
251 {
252     if (if_name != "port") {
253         return AbstractMemory::getPort(if_name, idx);
254     } else {
255         return port;
256     }
```

```
257 }
258
259
260 void NVMainMemory::NVMainStatPrinter::process()
261 {
262     assert(nvmainPtr != NULL);
263
264     assert(curTick() >= memory->lastWakeup);
265     Tick stepCycles = (curTick() - memory->lastWakeup) / memory->clock;
266
267     memory->m_nvmainGlobalEventQueue->Cycle( stepCycles );
268
269     nvmainPtr->CalculateStats();
270     std::ostream& refStream = (statStream.is_open()) ? statStream :
    ↪  std::cout;
271     nvmainPtr->GetStats()->PrintAll( refStream );
272 }
273
274
275 void NVMainMemory::NVMainStatReseter::process()
276 {
277     assert(nvmainPtr != NULL);
278
279     nvmainPtr->ResetStats();
280     nvmainPtr->GetStats()->ResetAll( );
281 }
282
283
284 NVMainMemory::MemoryPort::MemoryPort(const std::string& _name,
    ↪  NVMainMemory& _memory)
285     : ResponsePort(_name, &_memory), memory(_memory), forgdb(_memory)
286 {
287         // traceFile.open("nvmain_mem.trace", std::ios_base::out);
288 }
289
```

```
290
291 AddrRangeList NVMainMemory::MemoryPort::getAddrRanges() const
292 {
293     AddrRangeList ranges;
294     ranges.push_back(memory.getAddrRange());
295     return ranges;
296 }
297
298
299 void
300 NVMainMemory::SetRequestData(NVMainRequest *request, PacketPtr pkt)
301 {
302     uint8_t *hostAddr;
303
304     request->data.SetSize( pkt->getSize() );
305     request->oldData.SetSize( pkt->getSize() );
306
307     if (pkt->isRead())
308     {
309         const RequestPtr dataReq =
310                     std::make_shared<Request>(pkt->getAddr(),
                        ↪  pkt->getSize(), 0,
311                                                 Request::funcRequestorId);
312         Packet *dataPkt = new Packet(dataReq, MemCmd::ReadReq);
313         dataPkt->allocate();
314         doFunctionalAccess(dataPkt);
315
316         hostAddr = new uint8_t[ pkt->getSize() ];
317         memcpy( hostAddr, dataPkt->getPtr<uint8_t>(), pkt->getSize() );
318
319         for(int i = 0; i < pkt->getSize(); i++ )
320         {
321             request->oldData.SetByte(i, *(hostAddr + i));
322             request->data.SetByte(i, *(hostAddr + i));
323         }
```

```
324
325         delete dataPkt;
326         delete [] hostAddr;
327     }
328     else
329     {
330         const RequestPtr dataReq =
331                     std::make_shared<Request>(pkt->getAddr(),
                        ↪  pkt->getSize(), 0,
332                                                 Request::funcRequestorId);
333         Packet *dataPkt = new Packet(dataReq, MemCmd::ReadReq);
334         dataPkt->allocate();
335         doFunctionalAccess(dataPkt);
336
337         uint8_t *hostAddrT = new uint8_t[ pkt->getSize() ];
338         memcpy( hostAddrT, dataPkt->getPtr<uint8_t>(), pkt->getSize() );
339
340         hostAddr = new uint8_t[ pkt->getSize() ];
341         memcpy( hostAddr, pkt->getPtr<uint8_t>(), pkt->getSize() );
342
343         for(int i = 0; i < pkt->getSize(); i++ )
344         {
345             request->oldData.SetByte(i, *(hostAddrT + i));
346             request->data.SetByte(i, *(hostAddr + i));
347         }
348
349         delete dataPkt;
350         delete [] hostAddrT;
351         delete [] hostAddr;
352     }
353 }
354
355
356 Tick
357 NVMainMemory::MemoryPort::recvAtomic(PacketPtr pkt)
```

```
358 {
359     if (pkt->cacheResponding())
360         return 0;
361 
362     /*
363      * calculate the latency. Now it is only random number
364      */
365     Tick latency = memory.lat;
366 
367     if (memory.lat_var != 0)
368         latency += random_mt.random<Tick>(0, memory.lat_var);
369 
370     /*
371      *  if NVMain also needs the packet to warm up the inline cache, create
    ↪ the request
372      */
373     if( memory.NVMainWarmUp )
374     {
375         NVMainRequest *request = new NVMainRequest( );
376 
377         memory.SetRequestData( request, pkt );
378 
379         if( !pkt->isRead() && !pkt->isWrite() )
380         {
381             // if it is neither read nor write, just return
382             // well, speed may suffer a little bit...
383             return latency;
384         }
385 
386         /* initialize the request so that NVMain can correctly serve it */
387         request->access = UNKNOWN_ACCESS;
388         request->address.SetPhysicalAddress(pkt->req->getPaddr());
389         request->status = MEM_REQUEST_INCOMPLETE;
390         request->type = (pkt->isRead()) ? READ : WRITE;
391         request->owner = (NVMObject *)&memory;
```

```
392         if(pkt->req->hasPC()) request->programCounter = pkt->req->getPC();
393         if(pkt->req->hasContextId()) request->threadId =
            ↪  pkt->req->contextId();
394
395         /*
396          * Issue the request to NVMain as an atomic request
397          */
398         memory.masterInstance->m_nvmainPtr->IssueAtomic(request);
399
400         delete request;
401     }
402
403     /*
404      * do the memory access to get the read data and change the response
    ↪   tag
405      */
406     memory.access(pkt);
407
408     return latency;
409 }
410
411
412 void
413 NVMainMemory::MemoryPort::recvFunctional(PacketPtr pkt)
414 {
415     pkt->pushLabel(memory.name());
416
417     memory.doFunctionalAccess(pkt);
418
419     pkt->popLabel();
420 }
421
422
423 bool
424 NVMainMemory::MemoryPort::recvTimingReq(PacketPtr pkt)
```

```
425 {
426     /* added by Tao @ 01/24/2013, just copy the code from SimpleMemory */
427     /// @todo temporary hack to deal with memory corruption issues until
428     /// 4-phase transactions are complete
429     for (int x = 0; x < memory.pendingDelete.size(); x++)
430         delete memory.pendingDelete[x];
431     memory.pendingDelete.clear();
432 
433     if (pkt->cacheResponding()) {
434         memory.pendingDelete.push_back(pkt);
435         return true;
436     }
437 
438     /** Imprimir Tick **/
439     // traceFile << curTick() << ",";
440 
441     // // TimingSimpleCPU* instancia_derivada =
        ↪  dynamic_cast<TimingSimpleCPU*>(system->find("system.cpu"));
442     // // /** Imprimir Instr **/
443     // // traceFile << instancia_derivada->numSimulatedInsts() << ",";
444 
445     // /** ID paquete **/
446     // //traceFile << pkt->id << " ";
447 
448     /** Tipo de petición **/
449     // if (pkt->isRead())
450     //     traceFile << "R,";
451     // else
452     // {
453     //     if(pkt->isWrite())
454     //     {
455     //         if(pkt->isWriteback())
456     //             traceFile << "Y,";
457     //         else
458     //             traceFile << "N,";
```

```
459     //     }
460     //     else
461     //         if(pkt->isWriteback())
462     //             traceFile << "W,";
463     //         else
464     //             traceFile << "X,";
465     // }
466
467
468     // /** Direccion de memoria **/
469     // traceFile << std::hex << "0x" << pkt->getAddr() << std::dec << ",";
470
471     // /** Tamanyo de la peticion **/
472     // traceFile << pkt->getSize() << std::endl;
473
474     // traceFile.flush();
475
476     if (!pkt->isRead() && !pkt->isWrite()) {
477         DPRINTF(NVMain, "NVMainMemory: Received a packet that is neither
            ↪ read nor write.\n");
478         DPRINTF(NVMainMin, "NVMainMemory: Received a packet that is neither
            ↪ read nor write.\n");
479
480         bool needsResponse = pkt->needsResponse();
481
482         memory.access(pkt);
483         if (needsResponse) {
484             assert(pkt->isResponse());
485
486             pkt->headerDelay = pkt->payloadDelay = 0;
487
488             memory.responseQueue.push_back(pkt);
489
490             memory.ScheduleResponse( );
491         } else {
```

```
492             memory.pendingDelete.push_back(pkt);
493         }
494 
495         return true;
496     }
497 
498 
499     if (memory.retryRead || memory.retryWrite)
500     {
501         DPRINTF(NVMain, "nvmain_mem.cc: Received request while waiting for
        ↪  retry!\n");
502         DPRINTF(NVMainMin, "nvmain_mem.cc: Received request while waiting
        ↪  for retry!\n");
503         return false;
504     }
505 
506     /*
507     * If the difference between the number of ticks running in the system
508     * and the number of subsystems is within 5 cycles, they are considered
    ↪  to be synchronized.
509     * Otherwise, the clock must be synchronized.
510     */
511     if (memory.sync == false && memory.retryflag == false) {
512         ncycle_t stepCycles = (curTick() - memory.lastWakeup) /
        ↪  memory.clock;
513         memory.masterInstance->m_nvmainGlobalEventQueue->Cycle(stepCycles);
514     }
515     memory.lastWakeup = curTick();
516 
517     // Bus latency is modeled in NVMain.
518     pkt->headerDelay = pkt->payloadDelay = 0;
519 
520     NVMainRequest *request = new NVMainRequest( );
521 
522     bool canQueue, enqueued = false;
```

```
523
524     memory.SetRequestData( request, pkt );
525
526     /*
527      *  NVMain expects linear addresses, so hack: If we are not the master
528      *  instance, assume there are two channels because 3GB-4GB is skipped
529      *  in X86 and subtract 1GB.
530      *
531      *  TODO: Have each channel communicate it's address range to determine
532      *  this fix up value.
533      */
534     uint64_t addressFixUp = 0;
535 #if THE_ISA == X86_ISA
536     if( masterInstance != &memory )
537     {
538         addressFixUp = 0x40000000;
539     }
540 #elif THE_ISA == ARM_ISA
541     /*
542      *  ARM regions are 2GB - 4GB followed by 34 GB - 64 GB. Work for up to
543      *  34 GB of memory. Further regions from 512 GB - 992 GB.
544      */
545     addressFixUp = 0x80000000;
546 #endif
547
548     request->access = UNKNOWN_ACCESS;
549     request->address.SetPhysicalAddress(pkt->req->getPaddr() -
      ↪  addressFixUp);
550     request->status = MEM_REQUEST_INCOMPLETE;
551     request->type = (pkt->isRead()) ? READ : WRITE;
552     request->owner = (NVMObject *)&memory;
553     request->arrivalTick = curTick();
554
555     if(pkt->req->hasPC()) request->programCounter = pkt->req->getPC();
556     if(pkt->req->hasContextId()) request->threadId = pkt->req->contextId();
```

```
557
558     /* Call hooks here manually, since there is no one else to do it. */
559     std::vector<NVMObject *>& preHooks  = memory.masterInstance->GetHooks(
        ↪  NVMHOOK_PREISSUE );
560     std::vector<NVMObject *>& postHooks = memory.masterInstance->GetHooks(
        ↪  NVMHOOK_POSTISSUE );
561     std::vector<NVMObject *>::iterator it;
562
563     canQueue = memory.masterInstance->GetChild( )->IsIssuable( request );
564
565     if( canQueue )
566     {
567         /* Call pre-issue hooks */
568         for( it = preHooks.begin(); it != preHooks.end(); it++ )
569         {
570             (*it)->SetParent( memory.masterInstance );
571             (*it)->IssueCommand( request );
572         }
573
574         enqueued = memory.masterInstance->GetChild(
            ↪  )->IssueCommand(request);
575         assert( enqueued == true );
576
577         NVMainMemoryRequest *memRequest = new NVMainMemoryRequest;
578
579         memRequest->request = request;
580         memRequest->packet = pkt;
581         memRequest->issueTick = curTick();
582         memRequest->atomic = false;
583
584         DPRINTF(NVMain, "nvmain_mem.cc: Enqueued Mem request for 0x%x of
            ↪  type %s\n", request->address.GetPhysicalAddress( ),
            ↪  ((pkt->isRead()) ? "READ" : "WRITE") );
585
586         /* See if we need to reschedule the wakeup event sooner. */
```

```
587         ncycle_t nextEvent =
            ↪ memory.masterInstance->m_nvmainGlobalEventQueue->GetNextEvent(NULL);
588         DPRINTF(NVMain, "NVMainMemory: Next event after issue is %d\n",
            ↪ nextEvent);
589         if( nextEvent < memory.nextEventCycle &&
            ↪ masterInstance->clockEvent.scheduled() )
590         {
591             ncycle_t currentCycle =
                ↪ memory.masterInstance->m_nvmainGlobalEventQueue->GetCurrentCycle();
592
593             //assert(nextEvent >= currentCycle);
594             ncycle_t stepCycles;
595             if( nextEvent > currentCycle )
596                 stepCycles = nextEvent - currentCycle;
597             else
598                 stepCycles = 1;
599
600             Tick nextWake = curTick() + memory.clock *
                ↪ static_cast<Tick>(stepCycles);
601
602             DPRINTF(NVMain, "NVMainMemory: Next event: %d CurrentCycle:
                ↪ %d\n", nextEvent, currentCycle);
603             DPRINTF(NVMain, "NVMainMemory: Rescheduled wake at %d after %d
                ↪ cycles\n", nextWake, stepCycles);
604
605             memory.nextEventCycle = nextEvent;
606             memory.ScheduleClockEvent( nextWake );
607         }
608         else if( !masterInstance->clockEvent.scheduled() )
609         {
610             ncycle_t currentCycle =
                ↪ memory.masterInstance->m_nvmainGlobalEventQueue->GetCurrentCycle();
611
612             //assert(nextEvent >= currentCycle);
613             ncycle_t stepCycles = nextEvent - currentCycle;
```

```
614             if( stepCycles == 0 || nextEvent < currentCycle )
615                 stepCycles = 1;
616 
617             Tick nextWake = curTick() + memory.clock *
                ↪  static_cast<Tick>(stepCycles);
618 
619             memory.nextEventCycle = nextEvent;
620             memory.ScheduleClockEvent( nextWake );
621         }
622 
623         memory.masterInstance->m_request_map.insert(
                ↪  std::pair<NVMainRequest *, NVMainMemoryRequest *>( request,
                ↪  memRequest ) );
624         memory.m_requests_outstanding++;
625 
626         /*
627          *  It seems gem5 will block until the packet gets a response, so
    ↪  create a copy of the request, so
628          *  the memory controller has it, then delete the original copy to
    ↪  respond to the packet.
629          */
630         if( request->type == WRITE )
631         {
632             NVMainMemoryRequest *requestCopy = new NVMainMemoryRequest( );
633 
634             requestCopy->request = new NVMainRequest( );
635             *(requestCopy->request) = *request;
636             requestCopy->packet = pkt;
637             requestCopy->issueTick = curTick();
638             requestCopy->atomic = false;
639 
640             memRequest->packet = NULL;
641 
```

```
642             memory.masterInstance->m_request_map.insert(
                ↪  std::pair<NVMainRequest *, NVMainMemoryRequest *>(
                ↪  requestCopy->request, requestCopy ) );
643             memory.m_requests_outstanding++;
644 
645             memory.RequestComplete( requestCopy->request );
646         }
647 
648         /* Call post-issue hooks. */
649         if( request != NULL )
650         {
651             for( it = postHooks.begin(); it != postHooks.end(); it++ )
652             {
653                 (*it)->SetParent( &memory );
654                 (*it)->IssueCommand( request );
655             }
656         }
657     }
658     else
659     {
660         DPRINTF(NVMain, "nvmain_mem.cc: Can not enqueue Mem request for
                ↪  0x%x of type %s\n", request->address.GetPhysicalAddress( ),
                ↪  ((pkt->isRead()) ? "READ" : "WRITE") );
661         DPRINTF(NVMainMin, "nvmain_mem.cc: Can not enqueue Mem request for
                ↪  0x%x of type %s\n", request->address.GetPhysicalAddress( ),
                ↪  ((pkt->isRead()) ? "READ" : "WRITE") );
662 
663         if (pkt->isRead())
664         {
665             memory.retryRead = true;
666         }
667         else
668         {
669             memory.retryWrite = true;
670         }
```

```
671
672             delete request;
673             request = NULL;
674         }
675
676         return enqueued;
677     }
678
679
680
681     Tick NVMainMemory::doAtomicAccess(PacketPtr pkt)
682     {
683         access(pkt);
684         return static_cast<Tick>(m_avgAtomicLatency);
685     }
686
687
688
689     void NVMainMemory::doFunctionalAccess(PacketPtr pkt)
690     {
691         functionalAccess(pkt);
692     }
693
694
695     DrainState NVMainMemory::drain()
696     {
697         if( !masterInstance->m_request_map.empty() )
698         {
699             return DrainState::Draining;
700         }
701         else
702         {
703             return DrainState::Drained;
704         }
705     }
```

```
706
707
708 void NVMainMemory::MemoryPort::recvRespRetry( )
709 {
710     memory.recvRetry( );
711 }
712
713
714 void NVMainMemory::MemoryPort::recvRetry( )
715 {
716     memory.recvRetry( );
717 }
718
719
720 void NVMainMemory::recvRetry( )
721 {
722     DPRINTF(NVMain, "NVMainMemory: recvRetry() called.\n");
723     DPRINTF(NVMainMin, "NVMainMemory: recvRetry() called.\n");
724
725     retryResp = false;
726     SendResponses( );
727 }
728
729
730 bool NVMainMemory::RequestComplete(NVM::NVMainRequest *req)
731 {
732     bool isRead = (req->type == READ || req->type == READ_PRECHARGE);
733     bool isWrite = (req->type == WRITE || req->type == WRITE_PRECHARGE);
734
735     /* Ignore bus read/write requests generated by the banks. */
736     if( req->type == BUS_WRITE || req->type == BUS_READ )
737     {
738         delete req;
739         return true;
740     }
```

```
741
742     NVMainMemoryRequest *memRequest;
743     std::map<NVMainRequest *, NVMainMemoryRequest *>::iterator iter;
744
745     // Find the mem request pointer in the map.
746     assert(masterInstance->m_request_map.count(req) != 0);
747     iter = masterInstance->m_request_map.find(req);
748     memRequest = iter->second;
749
750     if(!memRequest->atomic)
751     {
752         bool respond = false;
753
754         NVMainMemory *ownerInstance = dynamic_cast<NVMainMemory *>(
            ↪  req->owner );
755         assert( ownerInstance != NULL );
756
757         if( memRequest->packet )
758         {
759             respond = memRequest->packet->needsResponse();
760             ownerInstance->access(memRequest->packet);
761         }
762
763         for( auto retryIter = masterInstance->allInstances.begin();
764              retryIter != masterInstance->allInstances.end(); retryIter++ )
765         {
766             (*retryIter)->retryflag = true;
767             if( (*retryIter)->retryRead && (isRead || isWrite) )
768             {
769                 (*retryIter)->retryRead = false;
770                 (*retryIter)->port.sendRetryReq();
771             }
772             if( (*retryIter)->retryWrite && (isRead || isWrite) )
773             {
774                 (*retryIter)->retryWrite = false;
```

```
775                 (*retryIter)->port.sendRetryReq();
776             }
777             (*retryIter)->retryflag = false;
778         }
779 
780         DPRINTF(NVMain, "Completed Mem request for 0x%x of type %s\n",
            ↪ req->address.GetPhysicalAddress( ), (isRead ? "READ" :
            ↪ "WRITE"));
781 
782         if(respond)
783         {
784             ownerInstance->responseQueue.push_back(memRequest->packet);
785             ownerInstance->ScheduleResponse( );
786 
787             delete req;
788             delete memRequest;
789         }
790         else
791         {
792             if( memRequest->packet )
793                 ownerInstance->pendingDelete.push_back(memRequest->packet);
794 
795             CheckDrainState( );
796 
797             delete req;
798             delete memRequest;
799         }
800     }
801     else
802     {
803         delete req;
804         delete memRequest;
805     }
806 
807 
```

```
808     masterInstance->m_request_map.erase(iter);
809     //assert(m_requests_outstanding > 0);
810     m_requests_outstanding--;
811 
812     return true;
813 }
814 
815 
816 void NVMainMemory::SendResponses( )
817 {
818     if( responseQueue.empty() || retryResp == true )
819         return;
820 
821 
822     bool success = port.sendTimingResp( responseQueue.front() );
823 
824     if( success )
825     {
826         DPRINTF(NVMain, "NVMainMemory: Sending response.\n");
827 
828         responseQueue.pop_front( );
829 
830         if( !responseQueue.empty( ) )
831             ScheduleResponse( );
832 
833         CheckDrainState( );
834     }
835     else
836     {
837         DPRINTF(NVMain, "NVMainMemory: Retrying response.\n");
838         DPRINTF(NVMainMin, "NVMainMemory: Retrying response.\n");
839 
840         retryResp = true;
841     }
842 }
```

```
843
844
845 void NVMainMemory::CheckDrainState( )
846 {
847     if( drainManager != NULL && masterInstance->m_request_map.empty() )
848     {
849         DPRINTF(NVMain, "NVMainMemory: Drain completed.\n");
850         DPRINTF(NVMainMin, "NVMainMemory: Drain completed.\n");
851
852         drainManager->signalDrainDone( );
853         drainManager = NULL;
854     }
855 }
856
857
858 void NVMainMemory::ScheduleResponse( )
859 {
860     if( !respondEvent.scheduled( ) )
861         schedule(respondEvent, curTick() + clock);
862 }
863
864
865 void NVMainMemory::ScheduleClockEvent( Tick nextWake )
866 {
867     if( !masterInstance->clockEvent.scheduled() )
868         schedule(masterInstance->clockEvent, nextWake);
869     else
870         reschedule(masterInstance->clockEvent, nextWake);
871 }
872
873
874 void NVMainMemory::serialize(CheckpointOut &cp) const
875 {
876     if (masterInstance != this)
877         return;
```

```
878
879     std::string nvmain_chkpt_dir = "";
880
881     if( m_nvmainConfig->KeyExists( "CheckpointDirectory" ) )
882         nvmain_chkpt_dir = m_nvmainConfig->GetString( "CheckpointDirectory"
            ↪ );
883
884     if( nvmain_chkpt_dir != "" )
885     {
886         std::cout << "NVMainMemory: Writing to checkpoint directory " <<
            ↪ nvmain_chkpt_dir << std::endl;
887
888         m_nvmainPtr->CreateCheckpoint( nvmain_chkpt_dir );
889     }
890 }
891
892
893 void NVMainMemory::unserialize(CheckpointIn &cp)
894 {
895     if (masterInstance != this)
896         return;
897
898     std::string nvmain_chkpt_dir = "";
899
900     if( m_nvmainConfig->KeyExists( "CheckpointDirectory" ) )
901         nvmain_chkpt_dir = m_nvmainConfig->GetString( "CheckpointDirectory"
            ↪ );
902
903     if( nvmain_chkpt_dir != "" )
904     {
905         std::cout << "NVMainMemory: Reading from checkpoint directory " <<
            ↪ nvmain_chkpt_dir << std::endl;
906
907         m_nvmainPtr->RestoreCheckpoint( nvmain_chkpt_dir );
908     }
```

```
909 }
910
911
912 void NVMainMemory::tick( )
913 {
914     // Cycle memory controller
915     if (masterInstance == this)
916     {
917         /* Keep NVMain in sync with gem5. */
918         sync = true;
919         assert(curTick() >= lastWakeup);
920         ncycle_t stepCycles = (curTick() - lastWakeup) / clock;
921
922         DPRINTF(NVMain, "NVMainMemory: Stepping %d cycles\n", stepCycles);
923         m_nvmainGlobalEventQueue->Cycle( stepCycles );
924
925         lastWakeup = curTick();
926
927         ncycle_t nextEvent;
928
929         nextEvent = m_nvmainGlobalEventQueue->GetNextEvent(NULL);
930         if( nextEvent != std::numeric_limits<ncycle_t>::max() )
931         {
932             ncycle_t currentCycle =
                 ↪ m_nvmainGlobalEventQueue->GetCurrentCycle();
933
934             assert(nextEvent >= currentCycle);
935             stepCycles = nextEvent - currentCycle;
936
937             Tick nextWake = curTick() + clock *
                 ↪ static_cast<Tick>(stepCycles);
938
939             DPRINTF(NVMain, "NVMainMemory: Next event: %d CurrentCycle:
                 ↪ %d\n", nextEvent, currentCycle);
```

```
940             DPRINTF(NVMain, "NVMainMemory: Schedule wake for %d\n",
                ↪  nextWake);
941
942             nextEventCycle = nextEvent;
943             ScheduleClockEvent( nextWake );
944         }
945         sync = false;
946     }
947 }
```

APÉNDICE B

Código fuente del fichero «nvmain_mem.cc»

```
/*
 * Copyright (c) 2012-2014 Pennsylvania State University
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met: redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer;
 * redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution;
 * neither the name of the copyright holders nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
```

```
22  * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
23  * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
24  * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
25  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
26  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
27  */
28
29 /*
30  *  This file is part of NVMain- A cycle accurate timing, bit-accurate
31  *  energy simulator for non-volatile memory. Originally developed by
32  *  Matt Poremba at the Pennsylvania State University.
33  *
34  *  Website: http://www.cse.psu.edu/~poremba/nvmain/
35  *  Email: mrp5060@psu.edu
36  *
37  *  ---------------------------------------------------------------------
38  *
39  *  If you use this software for publishable research, please include
40  *  the original NVMain paper in the citation list and mention the use
41  *  of NVMain.
42  *
43  */
44
45 #include "SimInterface/Gem5Interface/Gem5Interface.h"
46 #include "Simulators/gem5/nvmain_mem.hh"
47 #include "Utils/HookFactory.h"
48
49 #include "base/random.hh"
50 #include "base/statistics.hh"
51 #include "debug/NVMain.hh"
52 #include "debug/NVMainMin.hh"
53 #include "config/the_isa.hh"
54
55 using namespace NVM;
56
```

```
57 // This members are singleton values used to hold the main instance of
58 // NVMain and it's wake/sleep (i.e., timing/atomic) status. These are
59 // needed since NVMain assumes a contiguous address range while gem5
60 // ISAs generally do not. The multiple instances allow for the gem5
61 // AddrRanges to be used normally while this class remapped to NVMains
62 // contiguous region.
63 NVMainMemory *NVMainMemory::masterInstance = NULL;
64
65 NVMainMemory::NVMainMemory(const Params &p)
66     : AbstractMemory(p), clockEvent(this), respondEvent(this),
67       drainManager(NULL), lat(p.atomic_latency),
68       lat_var(p.atomic_variance), nvmain_atomic(p.atomic_mode),
69       NVMainWarmUp(p.NVMainWarmUp), port(name() + ".port", *this)
70 {
71     char *cfgparams;
72     char *cfgvalues;
73     char *cparam, *cvalue;
74
75     char *saveptr1, *saveptr2;
76
77     nextEventCycle = 0;
78
79     m_nvmainPtr = NULL;
80     m_nacked_requests = false;
81
82     m_nvmainConfigPath = p.config;
83
84     m_nvmainConfig = new Config( );
85
86     m_nvmainConfig->Read( m_nvmainConfigPath );
87     std::cout << "NVMainControl: Reading NVMain config file: " <<
       ↪  m_nvmainConfigPath << "." << std::endl;
88
89     clock = clockPeriod( );
90
```

```
91      m_avgAtomicLatency = 100.0f;
92      m_numAtomicAccesses = 0;
93
94      retryRead = false;
95      retryWrite = false;
96      retryResp = false;
97      sync = false;
98      retryflag = false;
99      m_requests_outstanding = 0;
100
101     /*
102      * Modified by Tao @ 01/22/2013
103      * multiple parameters can be manually specified
104      * please separate the parameters by comma ","
105      * For example,
106      *    configparams = tRCD,tCAS,tRP
107      *    configvalues = 8,8,8
108      */
109     cfgparams = (char *)p.configparams.c_str();
110     cfgvalues = (char *)p.configvalues.c_str();
111
112     for( cparam = strtok_r( cfgparams, ",", &saveptr1 ), cvalue = strtok_r(
        ↪  cfgvalues, ",", &saveptr2 )
113             ; (cparam && cvalue) ; cparam = strtok_r( NULL, ",", &saveptr1
                ↪  ), cvalue = strtok_r( NULL, ",", &saveptr2) )
114     {
115         std::cout << "NVMain: Overriding parameter `" << cparam << "' with
            ↪  `" << cvalue << "'" << std::endl;
116         m_nvmainConfig->SetValue( cparam, cvalue );
117     }
118
119   BusWidth = m_nvmainConfig->GetValue( "BusWidth" );
120   tBURST = m_nvmainConfig->GetValue( "tBURST" );
121   RATE = m_nvmainConfig->GetValue( "RATE" );
122
```

```
123     lastWakeup = curTick();
124     startWakeup = curTick();
125 }
126
127
128 NVMainMemory::~NVMainMemory()
129 {
130     std::cout << "NVMain dtor called" << std::endl;
131 }
132
133
134 void
135 NVMainMemory::init()
136 {
137     if (!port.isConnected()) {
138         fatal("NVMainMemory %s is unconnected!\n", name());
139     } else {
140         port.sendRangeChange();
141     }
142
143     if( masterInstance == NULL )
144     {
145         masterInstance = this;
146
147         m_nvmainPtr = new NVM::NVMain( );
148         m_statsPtr = new NVM::Stats( );
149         m_nvmainSimInterface = new NVM::Gem5Interface( );
150         m_nvmainEventQueue = new NVM::EventQueue( );
151         m_nvmainGlobalEventQueue = new NVM::GlobalEventQueue( );
152         m_tagGenerator = new NVM::TagGenerator( 1000 );
153
154         m_nvmainConfig->SetSimInterface( m_nvmainSimInterface );
155
156         statPrinter.nvmainPtr = m_nvmainPtr;
157         statReseter.nvmainPtr = m_nvmainPtr;
```

```
158
159         if( m_nvmainConfig->KeyExists( "StatsFile" ) )
160         {
161             statPrinter.statStream.open( m_nvmainConfig->GetString(
                ↪  "StatsFile" ).c_str(),
162                                          std::ofstream::out |
                                             ↪  std::ofstream::app );
163         }
164
165         statPrinter.memory = this;
166         statPrinter.forgdb = this;
167
168         //registerExitCallback( &statPrinter );
169         ::Stats::registerDumpCallback([this]() { statPrinter.process(); });
170         ::Stats::registerResetCallback([this]() { statReseter.process();
            ↪  });
171
172         SetEventQueue( m_nvmainEventQueue );
173         SetStats( m_statsPtr );
174         SetTagGenerator( m_tagGenerator );
175
176         m_nvmainGlobalEventQueue->SetFrequency( m_nvmainConfig->GetEnergy(
            ↪  "CPUFreq" ) * 1000000.0 );
177         SetGlobalEventQueue( m_nvmainGlobalEventQueue );
178
179         clock = static_cast<Tick>(round(1000000 /
            ↪  m_nvmainConfig->GetEnergy("CPUFreq")));
180         // TODO: Confirm global event queue frequency is the same as this
            ↪  SimObject's clock.
181
182         /*  Add any specified hooks */
183         std::vector<std::string>& hookList = m_nvmainConfig->GetHooks( );
184
185         for( size_t i = 0; i < hookList.size( ); i++ )
186         {
```

```
187             std::cout << "Creating hook " << hookList[i] << std::endl;
188
189             NVMObject *hook = HookFactory::CreateHook( hookList[i] );
190
191             if( hook != NULL )
192             {
193                 AddHook( hook );
194                 hook->SetParent( this );
195                 hook->Init( m_nvmainConfig );
196             }
197             else
198             {
199                 std::cout << "Warning: Could not create a hook named `"
200                     << hookList[i] << "'." << std::endl;
201             }
202         }
203
204         /* Setup child and parent modules. */
205         AddChild( m_nvmainPtr );
206         m_nvmainPtr->SetParent( this );
207         m_nvmainGlobalEventQueue->AddSystem( m_nvmainPtr, m_nvmainConfig );
208         m_nvmainPtr->SetConfig( m_nvmainConfig );
209
210         masterInstance->allInstances.push_back(this);
211     }
212     else
213     {
214         masterInstance->allInstances.push_back(this);
215         masterInstance->otherInstance = this;
216     }
217 }
218
219
220 void NVMainMemory::startup()
221 {
```

```
222     DPRINTF(NVMain, "NVMainMemory: startup() called.\n");
223     DPRINTF(NVMainMin, "NVMainMemory: startup() called.\n");
224 
225     /*
226      *  Schedule the initial event. Needed for warmup and timing mode.
227      *  If we are in atomic/fast-forward, wakeup will be disabled upon
228      *  the first atomic request receieved in recvAtomic().
229      */
230     if (!masterInstance->clockEvent.scheduled())
231         schedule(masterInstance->clockEvent, curTick() + clock);
232 
233     lastWakeup = curTick();
234     startWakeup = curTick();
235 }
236 
237 
238 void NVMainMemory::wakeup()
239 {
240     DPRINTF(NVMain, "NVMainMemory: wakeup() called.\n");
241     DPRINTF(NVMainMin, "NVMainMemory: wakeup() called.\n");
242 
243     schedule(masterInstance->clockEvent, clockEdge());
244 
245     lastWakeup = curTick();
246 }
247 
248 
249 Port &
250 NVMainMemory::getPort(const std::string& if_name, PortID idx)
251 {
252     if (if_name != "port") {
253         return AbstractMemory::getPort(if_name, idx);
254     } else {
255         return port;
256     }
```

```
257 }
258
259
260 void NVMainMemory::NVMainStatPrinter::process()
261 {
262     assert(nvmainPtr != NULL);
263
264     assert(curTick() >= memory->lastWakeup);
265     Tick stepCycles = (curTick() - memory->lastWakeup) / memory->clock;
266
267     memory->m_nvmainGlobalEventQueue->Cycle( stepCycles );
268
269     nvmainPtr->CalculateStats();
270     std::ostream& refStream = (statStream.is_open()) ? statStream :
    ↪  std::cout;
271     nvmainPtr->GetStats()->PrintAll( refStream );
272 }
273
274
275 void NVMainMemory::NVMainStatReseter::process()
276 {
277     assert(nvmainPtr != NULL);
278
279     nvmainPtr->ResetStats();
280     nvmainPtr->GetStats()->ResetAll( );
281 }
282
283
284 NVMainMemory::MemoryPort::MemoryPort(const std::string& _name,
    ↪  NVMainMemory& _memory)
285     : ResponsePort(_name, &_memory), memory(_memory), forgdb(_memory)
286 {
287         // traceFile.open("nvmain_mem.trace", std::ios_base::out);
288 }
289
```

```
290
291 AddrRangeList NVMainMemory::MemoryPort::getAddrRanges() const
292 {
293     AddrRangeList ranges;
294     ranges.push_back(memory.getAddrRange());
295     return ranges;
296 }
297
298
299 void
300 NVMainMemory::SetRequestData(NVMainRequest *request, PacketPtr pkt)
301 {
302     uint8_t *hostAddr;
303
304     request->data.SetSize( pkt->getSize() );
305     request->oldData.SetSize( pkt->getSize() );
306
307     if (pkt->isRead())
308     {
309         const RequestPtr dataReq =
310                     std::make_shared<Request>(pkt->getAddr(),
                        ↪  pkt->getSize(), 0,
311                                                 Request::funcRequestorId);
312         Packet *dataPkt = new Packet(dataReq, MemCmd::ReadReq);
313         dataPkt->allocate();
314         doFunctionalAccess(dataPkt);
315
316         hostAddr = new uint8_t[ pkt->getSize() ];
317         memcpy( hostAddr, dataPkt->getPtr<uint8_t>(), pkt->getSize() );
318
319         for(int i = 0; i < pkt->getSize(); i++ )
320         {
321             request->oldData.SetByte(i, *(hostAddr + i));
322             request->data.SetByte(i, *(hostAddr + i));
323         }
```

```
324
325         delete dataPkt;
326         delete [] hostAddr;
327     }
328     else
329     {
330         const RequestPtr dataReq =
331                     std::make_shared<Request>(pkt->getAddr(),
                        ↪  pkt->getSize(), 0,
332                                                 Request::funcRequestorId);
333         Packet *dataPkt = new Packet(dataReq, MemCmd::ReadReq);
334         dataPkt->allocate();
335         doFunctionalAccess(dataPkt);
336
337         uint8_t *hostAddrT = new uint8_t[ pkt->getSize() ];
338         memcpy( hostAddrT, dataPkt->getPtr<uint8_t>(), pkt->getSize() );
339
340         hostAddr = new uint8_t[ pkt->getSize() ];
341         memcpy( hostAddr, pkt->getPtr<uint8_t>(), pkt->getSize() );
342
343         for(int i = 0; i < pkt->getSize(); i++ )
344         {
345             request->oldData.SetByte(i, *(hostAddrT + i));
346             request->data.SetByte(i, *(hostAddr + i));
347         }
348
349         delete dataPkt;
350         delete [] hostAddrT;
351         delete [] hostAddr;
352     }
353 }
354
355
356 Tick
357 NVMainMemory::MemoryPort::recvAtomic(PacketPtr pkt)
```

```
358 {
359     if (pkt->cacheResponding())
360         return 0;
361 
362     /*
363      * calculate the latency. Now it is only random number
364      */
365     Tick latency = memory.lat;
366 
367     if (memory.lat_var != 0)
368         latency += random_mt.random<Tick>(0, memory.lat_var);
369 
370     /*
371      *  if NVMain also needs the packet to warm up the inline cache, create
    ↪ the request
372      */
373     if( memory.NVMainWarmUp )
374     {
375         NVMainRequest *request = new NVMainRequest( );
376 
377         memory.SetRequestData( request, pkt );
378 
379         if( !pkt->isRead() && !pkt->isWrite() )
380         {
381             // if it is neither read nor write, just return
382             // well, speed may suffer a little bit...
383             return latency;
384         }
385 
386         /* initialize the request so that NVMain can correctly serve it */
387         request->access = UNKNOWN_ACCESS;
388         request->address.SetPhysicalAddress(pkt->req->getPaddr());
389         request->status = MEM_REQUEST_INCOMPLETE;
390         request->type = (pkt->isRead()) ? READ : WRITE;
391         request->owner = (NVMObject *)&memory;
```

```
392         if(pkt->req->hasPC()) request->programCounter = pkt->req->getPC();
393         if(pkt->req->hasContextId()) request->threadId =
            ↪  pkt->req->contextId();
394
395         /*
396          * Issue the request to NVMain as an atomic request
397          */
398         memory.masterInstance->m_nvmainPtr->IssueAtomic(request);
399
400         delete request;
401     }
402
403     /*
404      * do the memory access to get the read data and change the response
    ↪   tag
405      */
406     memory.access(pkt);
407
408     return latency;
409 }
410
411
412 void
413 NVMainMemory::MemoryPort::recvFunctional(PacketPtr pkt)
414 {
415     pkt->pushLabel(memory.name());
416
417     memory.doFunctionalAccess(pkt);
418
419     pkt->popLabel();
420 }
421
422
423 bool
424 NVMainMemory::MemoryPort::recvTimingReq(PacketPtr pkt)
```

```
425 {
426     /* added by Tao @ 01/24/2013, just copy the code from SimpleMemory */
427     /// @todo temporary hack to deal with memory corruption issues until
428     /// 4-phase transactions are complete
429     for (int x = 0; x < memory.pendingDelete.size(); x++)
430         delete memory.pendingDelete[x];
431     memory.pendingDelete.clear();
432 
433     if (pkt->cacheResponding()) {
434         memory.pendingDelete.push_back(pkt);
435         return true;
436     }
437 
438     /** Imprimir Tick **/
439     // traceFile << curTick() << ",";
440 
441     // // TimingSimpleCPU* instancia_derivada =
        ↪  dynamic_cast<TimingSimpleCPU*>(system->find("system.cpu"));
442     // // /** Imprimir Instr **/
443     // // traceFile << instancia_derivada->numSimulatedInsts() << ",";
444 
445     // /** ID paquete **/
446     // //traceFile << pkt->id << " ";
447 
448     /** Tipo de petición **/
449     // if (pkt->isRead())
450     //     traceFile << "R,";
451     // else
452     // {
453     //     if(pkt->isWrite())
454     //     {
455     //         if(pkt->isWriteback())
456     //             traceFile << "Y,";
457     //         else
458     //             traceFile << "N,";
```

```
459     //     }
460     //     else
461     //         if(pkt->isWriteback())
462     //             traceFile << "W,";
463     //         else
464     //             traceFile << "X,";
465     // }
466
467
468     // /** Direccion de memoria **/
469     // traceFile << std::hex << "0x" << pkt->getAddr() << std::dec << ",";
470
471     // /** Tamanyo de la peticion **/
472     // traceFile << pkt->getSize() << std::endl;
473
474     // traceFile.flush();
475
476     if (!pkt->isRead() && !pkt->isWrite()) {
477         DPRINTF(NVMain, "NVMainMemory: Received a packet that is neither
            ↪ read nor write.\n");
478         DPRINTF(NVMainMin, "NVMainMemory: Received a packet that is neither
            ↪ read nor write.\n");
479
480         bool needsResponse = pkt->needsResponse();
481
482         memory.access(pkt);
483         if (needsResponse) {
484             assert(pkt->isResponse());
485
486             pkt->headerDelay = pkt->payloadDelay = 0;
487
488             memory.responseQueue.push_back(pkt);
489
490             memory.ScheduleResponse( );
491         } else {
```

```
492             memory.pendingDelete.push_back(pkt);
493         }
494 
495         return true;
496     }
497 
498 
499     if (memory.retryRead || memory.retryWrite)
500     {
501         DPRINTF(NVMain, "nvmain_mem.cc: Received request while waiting for
        ↪  retry!\n");
502         DPRINTF(NVMainMin, "nvmain_mem.cc: Received request while waiting
        ↪  for retry!\n");
503         return false;
504     }
505 
506     /*
507     * If the difference between the number of ticks running in the system
508     * and the number of subsystems is within 5 cycles, they are considered
    ↪  to be synchronized.
509     * Otherwise, the clock must be synchronized.
510     */
511     if (memory.sync == false && memory.retryflag == false) {
512         ncycle_t stepCycles = (curTick() - memory.lastWakeup) /
        ↪  memory.clock;
513         memory.masterInstance->m_nvmainGlobalEventQueue->Cycle(stepCycles);
514     }
515     memory.lastWakeup = curTick();
516 
517     // Bus latency is modeled in NVMain.
518     pkt->headerDelay = pkt->payloadDelay = 0;
519 
520     NVMainRequest *request = new NVMainRequest( );
521 
522     bool canQueue, enqueued = false;
```

```
523
524     memory.SetRequestData( request, pkt );
525
526     /*
527      *  NVMain expects linear addresses, so hack: If we are not the master
528      *  instance, assume there are two channels because 3GB-4GB is skipped
529      *  in X86 and subtract 1GB.
530      *
531      *  TODO: Have each channel communicate it's address range to determine
532      *  this fix up value.
533      */
534     uint64_t addressFixUp = 0;
535 #if THE_ISA == X86_ISA
536     if( masterInstance != &memory )
537     {
538         addressFixUp = 0x40000000;
539     }
540 #elif THE_ISA == ARM_ISA
541     /*
542      *  ARM regions are 2GB - 4GB followed by 34 GB - 64 GB. Work for up to
543      *  34 GB of memory. Further regions from 512 GB - 992 GB.
544      */
545     addressFixUp = 0x80000000;
546 #endif
547
548     request->access = UNKNOWN_ACCESS;
549     request->address.SetPhysicalAddress(pkt->req->getPaddr() -
      ↪  addressFixUp);
550     request->status = MEM_REQUEST_INCOMPLETE;
551     request->type = (pkt->isRead()) ? READ : WRITE;
552     request->owner = (NVMObject *)&memory;
553     request->arrivalTick = curTick();
554
555     if(pkt->req->hasPC()) request->programCounter = pkt->req->getPC();
556     if(pkt->req->hasContextId()) request->threadId = pkt->req->contextId();
```

```
557
558     /* Call hooks here manually, since there is no one else to do it. */
559     std::vector<NVMObject *>& preHooks  = memory.masterInstance->GetHooks(
        ↪  NVMHOOK_PREISSUE );
560     std::vector<NVMObject *>& postHooks = memory.masterInstance->GetHooks(
        ↪  NVMHOOK_POSTISSUE );
561     std::vector<NVMObject *>::iterator it;
562
563     canQueue = memory.masterInstance->GetChild( )->IsIssuable( request );
564
565     if( canQueue )
566     {
567         /* Call pre-issue hooks */
568         for( it = preHooks.begin(); it != preHooks.end(); it++ )
569         {
570             (*it)->SetParent( memory.masterInstance );
571             (*it)->IssueCommand( request );
572         }
573
574         enqueued = memory.masterInstance->GetChild(
            ↪  )->IssueCommand(request);
575         assert( enqueued == true );
576
577         NVMainMemoryRequest *memRequest = new NVMainMemoryRequest;
578
579         memRequest->request = request;
580         memRequest->packet = pkt;
581         memRequest->issueTick = curTick();
582         memRequest->atomic = false;
583
584         DPRINTF(NVMain, "nvmain_mem.cc: Enqueued Mem request for 0x%x of
            ↪  type %s\n", request->address.GetPhysicalAddress( ),
            ↪  ((pkt->isRead()) ? "READ" : "WRITE") );
585
586         /* See if we need to reschedule the wakeup event sooner. */
```

```
587         ncycle_t nextEvent =
            ↪ memory.masterInstance->m_nvmainGlobalEventQueue->GetNextEvent(NULL);
588         DPRINTF(NVMain, "NVMainMemory: Next event after issue is %d\n",
            ↪ nextEvent);
589         if( nextEvent < memory.nextEventCycle &&
            ↪ masterInstance->clockEvent.scheduled() )
590         {
591             ncycle_t currentCycle =
                ↪ memory.masterInstance->m_nvmainGlobalEventQueue->GetCurrentCycle();
592
593             //assert(nextEvent >= currentCycle);
594             ncycle_t stepCycles;
595             if( nextEvent > currentCycle )
596                 stepCycles = nextEvent - currentCycle;
597             else
598                 stepCycles = 1;
599
600             Tick nextWake = curTick() + memory.clock *
                ↪ static_cast<Tick>(stepCycles);
601
602             DPRINTF(NVMain, "NVMainMemory: Next event: %d CurrentCycle:
                ↪ %d\n", nextEvent, currentCycle);
603             DPRINTF(NVMain, "NVMainMemory: Rescheduled wake at %d after %d
                ↪ cycles\n", nextWake, stepCycles);
604
605             memory.nextEventCycle = nextEvent;
606             memory.ScheduleClockEvent( nextWake );
607         }
608         else if( !masterInstance->clockEvent.scheduled() )
609         {
610             ncycle_t currentCycle =
                ↪ memory.masterInstance->m_nvmainGlobalEventQueue->GetCurrentCycle();
611
612             //assert(nextEvent >= currentCycle);
613             ncycle_t stepCycles = nextEvent - currentCycle;
```

```
614             if( stepCycles == 0 || nextEvent < currentCycle )
615                 stepCycles = 1;
616 
617             Tick nextWake = curTick() + memory.clock *
                ↪  static_cast<Tick>(stepCycles);
618 
619             memory.nextEventCycle = nextEvent;
620             memory.ScheduleClockEvent( nextWake );
621         }
622 
623         memory.masterInstance->m_request_map.insert(
                ↪  std::pair<NVMainRequest *, NVMainMemoryRequest *>( request,
                ↪  memRequest ) );
624         memory.m_requests_outstanding++;
625 
626         /*
627          *  It seems gem5 will block until the packet gets a response, so
    ↪  create a copy of the request, so
628          *  the memory controller has it, then delete the original copy to
    ↪  respond to the packet.
629          */
630         if( request->type == WRITE )
631         {
632             NVMainMemoryRequest *requestCopy = new NVMainMemoryRequest( );
633 
634             requestCopy->request = new NVMainRequest( );
635             *(requestCopy->request) = *request;
636             requestCopy->packet = pkt;
637             requestCopy->issueTick = curTick();
638             requestCopy->atomic = false;
639 
640             memRequest->packet = NULL;
641 
```

```
642             memory.masterInstance->m_request_map.insert(
                ↪  std::pair<NVMainRequest *, NVMainMemoryRequest *>(
                ↪  requestCopy->request, requestCopy ) );
643             memory.m_requests_outstanding++;
644
645             memory.RequestComplete( requestCopy->request );
646         }
647
648         /* Call post-issue hooks. */
649         if( request != NULL )
650         {
651             for( it = postHooks.begin(); it != postHooks.end(); it++ )
652             {
653                 (*it)->SetParent( &memory );
654                 (*it)->IssueCommand( request );
655             }
656         }
657     }
658     else
659     {
660         DPRINTF(NVMain, "nvmain_mem.cc: Can not enqueue Mem request for
                ↪  0x%x of type %s\n", request->address.GetPhysicalAddress( ),
                ↪  ((pkt->isRead()) ? "READ" : "WRITE") );
661         DPRINTF(NVMainMin, "nvmain_mem.cc: Can not enqueue Mem request for
                ↪  0x%x of type %s\n", request->address.GetPhysicalAddress( ),
                ↪  ((pkt->isRead()) ? "READ" : "WRITE") );
662
663         if (pkt->isRead())
664         {
665             memory.retryRead = true;
666         }
667         else
668         {
669             memory.retryWrite = true;
670         }
```

```
671
672             delete request;
673             request = NULL;
674         }
675
676         return enqueued;
677     }
678
679
680
681     Tick NVMainMemory::doAtomicAccess(PacketPtr pkt)
682     {
683         access(pkt);
684         return static_cast<Tick>(m_avgAtomicLatency);
685     }
686
687
688
689     void NVMainMemory::doFunctionalAccess(PacketPtr pkt)
690     {
691         functionalAccess(pkt);
692     }
693
694
695     DrainState NVMainMemory::drain()
696     {
697         if( !masterInstance->m_request_map.empty() )
698         {
699             return DrainState::Draining;
700         }
701         else
702         {
703             return DrainState::Drained;
704         }
705     }
```

```
706
707
708 void NVMainMemory::MemoryPort::recvRespRetry( )
709 {
710     memory.recvRetry( );
711 }
712
713
714 void NVMainMemory::MemoryPort::recvRetry( )
715 {
716     memory.recvRetry( );
717 }
718
719
720 void NVMainMemory::recvRetry( )
721 {
722     DPRINTF(NVMain, "NVMainMemory: recvRetry() called.\n");
723     DPRINTF(NVMainMin, "NVMainMemory: recvRetry() called.\n");
724
725     retryResp = false;
726     SendResponses( );
727 }
728
729
730 bool NVMainMemory::RequestComplete(NVM::NVMainRequest *req)
731 {
732     bool isRead = (req->type == READ || req->type == READ_PRECHARGE);
733     bool isWrite = (req->type == WRITE || req->type == WRITE_PRECHARGE);
734
735     /* Ignore bus read/write requests generated by the banks. */
736     if( req->type == BUS_WRITE || req->type == BUS_READ )
737     {
738         delete req;
739         return true;
740     }
```

```
741
742     NVMainMemoryRequest *memRequest;
743     std::map<NVMainRequest *, NVMainMemoryRequest *>::iterator iter;
744
745     // Find the mem request pointer in the map.
746     assert(masterInstance->m_request_map.count(req) != 0);
747     iter = masterInstance->m_request_map.find(req);
748     memRequest = iter->second;
749
750     if(!memRequest->atomic)
751     {
752         bool respond = false;
753
754         NVMainMemory *ownerInstance = dynamic_cast<NVMainMemory *>(
            ↪ req->owner );
755         assert( ownerInstance != NULL );
756
757         if( memRequest->packet )
758         {
759             respond = memRequest->packet->needsResponse();
760             ownerInstance->access(memRequest->packet);
761         }
762
763         for( auto retryIter = masterInstance->allInstances.begin();
764              retryIter != masterInstance->allInstances.end(); retryIter++ )
765         {
766             (*retryIter)->retryflag = true;
767             if( (*retryIter)->retryRead && (isRead || isWrite) )
768             {
769                 (*retryIter)->retryRead = false;
770                 (*retryIter)->port.sendRetryReq();
771             }
772             if( (*retryIter)->retryWrite && (isRead || isWrite) )
773             {
774                 (*retryIter)->retryWrite = false;
```

```
775                 (*retryIter)->port.sendRetryReq();
776             }
777             (*retryIter)->retryflag = false;
778         }
779 
780         DPRINTF(NVMain, "Completed Mem request for 0x%x of type %s\n",
            ↪ req->address.GetPhysicalAddress( ), (isRead ? "READ" :
            ↪ "WRITE"));
781 
782         if(respond)
783         {
784             ownerInstance->responseQueue.push_back(memRequest->packet);
785             ownerInstance->ScheduleResponse( );
786 
787             delete req;
788             delete memRequest;
789         }
790         else
791         {
792             if( memRequest->packet )
793                 ownerInstance->pendingDelete.push_back(memRequest->packet);
794 
795             CheckDrainState( );
796 
797             delete req;
798             delete memRequest;
799         }
800     }
801     else
802     {
803         delete req;
804         delete memRequest;
805     }
806 
807 
```

```
808     masterInstance->m_request_map.erase(iter);
809     //assert(m_requests_outstanding > 0);
810     m_requests_outstanding--;
811 
812     return true;
813 }
814 
815 
816 void NVMainMemory::SendResponses( )
817 {
818     if( responseQueue.empty() || retryResp == true )
819         return;
820 
821 
822     bool success = port.sendTimingResp( responseQueue.front() );
823 
824     if( success )
825     {
826         DPRINTF(NVMain, "NVMainMemory: Sending response.\n");
827 
828         responseQueue.pop_front( );
829 
830         if( !responseQueue.empty( ) )
831             ScheduleResponse( );
832 
833         CheckDrainState( );
834     }
835     else
836     {
837         DPRINTF(NVMain, "NVMainMemory: Retrying response.\n");
838         DPRINTF(NVMainMin, "NVMainMemory: Retrying response.\n");
839 
840         retryResp = true;
841     }
842 }
```

```
843
844
845 void NVMainMemory::CheckDrainState( )
846 {
847     if( drainManager != NULL && masterInstance->m_request_map.empty() )
848     {
849         DPRINTF(NVMain, "NVMainMemory: Drain completed.\n");
850         DPRINTF(NVMainMin, "NVMainMemory: Drain completed.\n");
851
852         drainManager->signalDrainDone( );
853         drainManager = NULL;
854     }
855 }
856
857
858 void NVMainMemory::ScheduleResponse( )
859 {
860     if( !respondEvent.scheduled( ) )
861         schedule(respondEvent, curTick() + clock);
862 }
863
864
865 void NVMainMemory::ScheduleClockEvent( Tick nextWake )
866 {
867     if( !masterInstance->clockEvent.scheduled() )
868         schedule(masterInstance->clockEvent, nextWake);
869     else
870         reschedule(masterInstance->clockEvent, nextWake);
871 }
872
873
874 void NVMainMemory::serialize(CheckpointOut &cp) const
875 {
876     if (masterInstance != this)
877         return;
```

```
878
879     std::string nvmain_chkpt_dir = "";
880
881     if( m_nvmainConfig->KeyExists( "CheckpointDirectory" ) )
882         nvmain_chkpt_dir = m_nvmainConfig->GetString( "CheckpointDirectory"
            ↪ );
883
884     if( nvmain_chkpt_dir != "" )
885     {
886         std::cout << "NVMainMemory: Writing to checkpoint directory " <<
            ↪ nvmain_chkpt_dir << std::endl;
887
888         m_nvmainPtr->CreateCheckpoint( nvmain_chkpt_dir );
889     }
890 }
891
892
893 void NVMainMemory::unserialize(CheckpointIn &cp)
894 {
895     if (masterInstance != this)
896         return;
897
898     std::string nvmain_chkpt_dir = "";
899
900     if( m_nvmainConfig->KeyExists( "CheckpointDirectory" ) )
901         nvmain_chkpt_dir = m_nvmainConfig->GetString( "CheckpointDirectory"
            ↪ );
902
903     if( nvmain_chkpt_dir != "" )
904     {
905         std::cout << "NVMainMemory: Reading from checkpoint directory " <<
            ↪ nvmain_chkpt_dir << std::endl;
906
907         m_nvmainPtr->RestoreCheckpoint( nvmain_chkpt_dir );
908     }
```

```
909 }
910
911
912 void NVMainMemory::tick( )
913 {
914     // Cycle memory controller
915     if (masterInstance == this)
916     {
917         /* Keep NVMain in sync with gem5. */
918         sync = true;
919         assert(curTick() >= lastWakeup);
920         ncycle_t stepCycles = (curTick() - lastWakeup) / clock;
921
922         DPRINTF(NVMain, "NVMainMemory: Stepping %d cycles\n", stepCycles);
923         m_nvmainGlobalEventQueue->Cycle( stepCycles );
924
925         lastWakeup = curTick();
926
927         ncycle_t nextEvent;
928
929         nextEvent = m_nvmainGlobalEventQueue->GetNextEvent(NULL);
930         if( nextEvent != std::numeric_limits<ncycle_t>::max() )
931         {
932             ncycle_t currentCycle =
                 ↪ m_nvmainGlobalEventQueue->GetCurrentCycle();
933
934             assert(nextEvent >= currentCycle);
935             stepCycles = nextEvent - currentCycle;
936
937             Tick nextWake = curTick() + clock *
                 ↪ static_cast<Tick>(stepCycles);
938
939             DPRINTF(NVMain, "NVMainMemory: Next event: %d CurrentCycle:
                 ↪ %d\n", nextEvent, currentCycle);
```

```
940             DPRINTF(NVMain, "NVMainMemory: Schedule wake for %d\n",
                ↪  nextWake);
941
942             nextEventCycle = nextEvent;
943             ScheduleClockEvent( nextWake );
944         }
945         sync = false;
946     }
947 }
```

APÉNDICE C

Fichero de configuración de ejemplo para CACTI

APÉNDICE D

Código fuente para la función «IsIssuable(...)» del fichero «nvmain.cpp»

```
1  if (!(request->flags & (NVMainRequest::FLAG_EVICT |
   ↪  NVMainRequest::FLAG_SKIP_TAG)) && useTagCache)
2  {
3      ++totalReqs;
4      channel = 1;
5      request->address.SetPhysicalAddress(GetDecoder()->ReverseTranslate(row,
       ↪  col, bank, rank, channel, subarray));
6      request->address.SetTranslatedAddress(row, col, bank, rank, channel,
       ↪  subarray);
7      int present = 0;
8      uint64_t cachedAddr = replacementPolicy->get(
       ↪  request->address.GetPhysicalAddress(), &present);
9      GetDecoder()->Translate(cachedAddr, &cRow, &cCol, &cBank, &cRank,
       ↪  &cChannel, &cSubarray);
10     if (!present) // Need to insert the new data in the DRAM cache
11     {
12         ++reqsMiss;
13         if (request->type == NVM::WRITE)
14         {
```

```
15              ++WRMiss;
16              if (replacementPolicy->isFull()) // When the cache has no more
                ↪  free addresses, a Writeback to NVRAM is needed
17              {
18                  ++tagReplacements;
19                  uint64_t freeAddr = replacementPolicy->removeLRU();
20
21                  replacementPolicy->put(
                    ↪  request->address.GetPhysicalAddress(), freeAddr);
22                  inverseMap[freeAddr] =
                    ↪  request->address.GetPhysicalAddress();
23
24                  // Channel should already be 0
25                  GetDecoder()->Translate(freeAddr, &row, &col, &bank, &rank,
                    ↪  &channel, &subarray);
26
27                  // Create DRAM Write request to the appropiate address
28                  NVMainRequest *DRAMWrite = new NVMainRequest();
29                  DRAMWrite->access = UNKNOWN_ACCESS;
30                  DRAMWrite->address.SetPhysicalAddress(freeAddr);
31                  DRAMWrite->address.SetTranslatedAddress(row, col, bank,
                    ↪  rank, channel, subarray);
32                  DRAMWrite->status = MEM_REQUEST_INCOMPLETE;
33                  DRAMWrite->type = NVM::WRITE;
34                  DRAMWrite->owner = this;
35                  DRAMWrite->data = request->data; // Data to be written
36                  DRAMWrite->flags |= NVMainRequest::FLAG_EVICT;
37                  DRAMWrite->flags |= NVMainRequest::FLAG_SKIP_TAG;
38                  waitingDRAMWrite.push_back(DRAMWrite);
39
40                  DRAMWriteSize++;
41
42                  if (DRAMWriteSize == 20)
43                      DRAMWriteSize--;
44
```

```
45              // Generate DRAM Read request to the freeAddr Address
46              NVMainRequest *DRead = new NVMainRequest();
47              DRead->access = UNKNOWN_ACCESS;
48              DRead->address.SetPhysicalAddress(freeAddr);
49              DRead->address.SetTranslatedAddress(row, col, bank, rank,
                ↪  channel, subarray);
50              DRead->status = MEM_REQUEST_INCOMPLETE;
51              DRead->type = READ;
52              DRead->owner = this;
53              DRead->flags |= NVMainRequest::FLAG_EVICT;
54              DRead->data.SetSize(64);
55
56              // Try to enqueue the new request, but first mark the
                ↪  request as waiting for DRAM Read
57              DRAMReadSize++;
58              if (DRAMReadSize == 20)
59                  DRAMReadSize--;
60              this->IssueCommand(DRead);
61
62              // Disregard the current request
63              return true;
64          }
65          else // Allocate one memory address
66          {
67              ncounter_t rRow, rCol, rBank, rRank, rChannel, rSubarray;
68              GetDecoder()->Translate(tmpAddr, &rRow, &rCol, &rBank,
                ↪  &rRank, &rChannel, &rSubarray);
69              rChannel = 0;
70              uint64_t randAddr = GetDecoder()->ReverseTranslate(rRow,
                ↪  rCol, rBank, rRank, rChannel, rSubarray);
71
72              while (replacementPolicy->has(randAddr))
73              {
74                  tmpAddr+=64;
```

```
75                  GetDecoder()->Translate(tmpAddr, &rRow, &rCol, &rBank,
                     ↪  &rRank, &rChannel, &rSubarray);
76                  rChannel = 0;
77                  randAddr = GetDecoder()->ReverseTranslate(rRow, rCol,
                     ↪  rBank, rRank, rChannel, rSubarray);
78              }
79              tmpAddr+=64;
80              row = rRow;
81              col = rCol;
82              bank = rBank;
83              rank = rRank;
84              channel = rChannel;
85              subarray = rSubarray;
86
87              replacementPolicy->put(
                 ↪  request->address.GetPhysicalAddress(), randAddr);
88              inverseMap.emplace(randAddr,
                 ↪  request->address.GetPhysicalAddress());
89
90              // Update the request target address
91              request->address.SetTranslatedAddress(row, col, bank, rank,
                 ↪  channel, subarray);
92              request->address.SetPhysicalAddress(GetDecoder()->
                 ↪  ReverseTranslate(row, col, bank, rank, channel,
                 ↪  subarray));
93          }
94      }
95      else
96      {
97          ++RDMiss;
98          // Update the request target address
99          request->address.SetTranslatedAddress(row, col, bank, rank,
             ↪  channel, subarray);
100         request->address.SetPhysicalAddress(GetDecoder()->
             ↪  ReverseTranslate(row, col, bank, rank, channel, subarray));
```

```
101             }
102         }
103         else // Data already present in DRAM cache, read address.
104         {
105             ++reqsHit;
106             GetDecoder()->Translate(cachedAddr, &row, &col,
107                                     &bank, &rank, &channel, &subarray);
108 
109             if (request->type == NVM::READ)
110                 ++RDHit;
111             else
112             {
113                 if (request->type == NVM::WRITE)
114                     ++WRHit;
115             }
116             // Update the request target address
117             request->address.SetTranslatedAddress(row, col, bank, rank,
                ↪  channel, subarray);
118             request->address.SetPhysicalAddress(GetDecoder()->
                ↪  ReverseTranslate(row, col, bank, rank, channel, subarray));
119         }
120     }
```

APÉNDICE E

Código fuente para la función «RequestComplete(...)» del fichero «nvmain.cpp»

```
1  if ((request->type == NVM::READ) && useTagCache && !(request->flags &
   ↪  NVMainRequest::FLAG_EVICT) &&
2      !(request->flags & NVMainRequest::FLAG_SKIP_TAG) &&
       ↪  !(request->isPrefetch))
3  {
4      ++readsOps;
5      uint64_t row, col, bank, rank, channel, subarray;
6      uint64_t cRow, cCol, cBank, cRank, cChannel, cSubarray;
7      GetDecoder()->Translate(request->address.GetPhysicalAddress(), &row,
       ↪  &col, &bank, &rank, &channel, &subarray);
8      if (channel != 0)
9      {
10         ++readsNotDRAM;
11         request->address.SetPhysicalAddress(GetDecoder()->
           ↪  ReverseTranslate(row, col, bank, rank, channel, subarray));
12         int present = 0;
13         uint64_t cachedAddr = replacementPolicy->get(request->
           ↪  address.GetPhysicalAddress(), &present);
```

```
14          GetDecoder()->Translate(cachedAddr, &cRow, &cCol, &cBank, &cRank,
            ↪  &cChannel, &cSubarray);
15          if (!present) // Need to insert the new data in the DRAM cache
16          {
17              if (replacementPolicy->isFull()) // When the cache has no more
                ↪  free addresses, a Writeback to NVRAM is needed
18              {
19                  ++RD_Replc;
20                  uint64_t freeAddr = replacementPolicy->removeLRU();
21
22
                    ↪  replacementPolicy->put(request->address.GetPhysicalAddress(),
                    ↪  freeAddr);
23                  inverseMap[freeAddr] =
                    ↪  request->address.GetPhysicalAddress();
24
25                  // Channel should already be 0
26                  GetDecoder()->Translate(freeAddr, &row, &col, &bank, &rank,
                    ↪  &channel, &subarray);
27
28                  // Create DRAM Write request to the appropiate address
29                  NVMainRequest *DRAMWrite = new NVMainRequest();
30                  DRAMWrite->access = UNKNOWN_ACCESS;
31                  DRAMWrite->address.SetPhysicalAddress(freeAddr);
32                  DRAMWrite->address.SetTranslatedAddress(row, col, bank,
                    ↪  rank, channel, subarray);
33                  DRAMWrite->status = MEM_REQUEST_INCOMPLETE;
34                  DRAMWrite->type = NVM::WRITE;
35                  DRAMWrite->owner = this;
36                  DRAMWrite->data = request->data; // Data to be written
37                  DRAMWrite->flags |= NVMainRequest::FLAG_EVICT;
38                  DRAMWrite->flags |= NVMainRequest::FLAG_SKIP_TAG;
39                  waitingDRAMWrite.push_back(DRAMWrite);
40
41                  DRAMWriteSize++;
```

```
42
43                  if (DRAMWriteSize == 20)
44                      DRAMWriteSize--;
45
46                  // Generate DRAM Read request to the freeAddr Address
47                  NVMainRequest *DRead = new NVMainRequest();
48                  DRead->access = UNKNOWN_ACCESS;
49                  DRead->address.SetPhysicalAddress(freeAddr);
50                  DRead->address.SetTranslatedAddress(row, col, bank, rank,
                    ↪  channel, subarray);
51                  DRead->status = MEM_REQUEST_INCOMPLETE;
52                  DRead->type = NVM::READ;
53                  DRead->owner = this;
54                  DRead->flags |= NVMainRequest::FLAG_EVICT;
55                  DRead->data.SetSize(64);
56
57                  // Try to enqueue the new request, but first mark the
                    ↪  request as waiting for DRAM Read
58                  DRAMReadSize++;
59
60                  if (DRAMReadSize == 20)
61                      DRAMReadSize--;
62                  this->IssueCommand(DRead);
63              }
64              else // Allocate one memory address
65              {
66                  ++RD_Clean;
67                  ncounter_t rRow, rCol, rBank, rRank, rChannel, rSubarray;
68                  GetDecoder()->Translate(tmpAddr, &rRow, &rCol, &rBank,
                    ↪  &rRank, &rChannel, &rSubarray);
69                  rChannel = 0;
70                  uint64_t randAddr = GetDecoder()->ReverseTranslate(rRow,
                    ↪  rCol, rBank, rRank, rChannel, rSubarray);
71
72                  while (replacementPolicy->has(randAddr))
```

```
73                  {
74                      tmpAddr+=64;
75                      GetDecoder()->Translate(tmpAddr, &rRow, &rCol, &rBank,
                        ↪  &rRank, &rChannel, &rSubarray);
76                      rChannel = 0;
77                      randAddr = GetDecoder()->ReverseTranslate(rRow, rCol,
                        ↪  rBank, rRank, rChannel, rSubarray);
78                  }
79                  tmpAddr+=64;
80                  row = rRow;
81                  col = rCol;
82                  bank = rBank;
83                  rank = rRank;
84                  channel = rChannel;
85                  subarray = rSubarray;
86
87                  replacementPolicy->put(request->
                    ↪  address.GetPhysicalAddress(), randAddr);
88                  inverseMap.emplace(randAddr,
                    ↪  request->address.GetPhysicalAddress());
89
90                  // Send the request to DRAM directly
91                  NVMainRequest *DRAMWrite = new NVMainRequest();
92                  DRAMWrite->access = UNKNOWN_ACCESS;
93                  DRAMWrite->address.SetPhysicalAddress(randAddr);
94                  DRAMWrite->address.SetTranslatedAddress(row, col, bank,
                    ↪  rank, channel, subarray);
95                  DRAMWrite->status = MEM_REQUEST_INCOMPLETE;
96                  DRAMWrite->type = NVM::WRITE;
97                  DRAMWrite->owner = this;
98                  DRAMWrite->data = request->data; // Data to be written
99                  DRAMWrite->flags |= NVMainRequest::FLAG_EVICT;
100                 DRAMWrite->flags |= NVMainRequest::FLAG_SKIP_TAG;
101
102                 DRAMWriteSize++;
```

```
103
104
105                 if (DRAMWriteSize == 20)
106                     DRAMWriteSize--;
107
108                 this->IssueCommand(DRAMWrite);
109             }
110         }
111         else // Data already present in DRAM cache, write to that address.
112         {
113             ++RD_Present;
114             GetDecoder()->Translate(cachedAddr, &row, &col,
115                                     &bank, &rank, &channel, &subarray);
116
117             // Send the request to DRAM directly
118             NVMainRequest *DRAMWrite = new NVMainRequest();
119             DRAMWrite->access = UNKNOWN_ACCESS;
120             DRAMWrite->address.SetPhysicalAddress(GetDecoder()->
                ↪ ReverseTranslate(row, col, bank, rank, channel, subarray));
121             DRAMWrite->address.SetTranslatedAddress(row, col, bank, rank,
                ↪ channel, subarray);
122             DRAMWrite->status = MEM_REQUEST_INCOMPLETE;
123             DRAMWrite->type = NVM::WRITE;
124             DRAMWrite->owner = this;
125             DRAMWrite->data = request->data; // Data to be written
126             DRAMWrite->flags |= NVMainRequest::FLAG_EVICT;
127             DRAMWrite->flags |= NVMainRequest::FLAG_SKIP_TAG;
128
129             DRAMWriteSize++;
130
131
132             if (DRAMWriteSize == 20)
133                 DRAMWriteSize--;
134
135             this->IssueCommand(DRAMWrite);
```

```
136             }
137         }
138     }
139
140     if (request->owner == this) // All Requests owners should be _this_
141     {
142         if (request->isPrefetch)
143         {
144             /* Place in prefetch buffer. */
145             if (prefetchBuffer.size() >= p->PrefetchBufferSize)
146             {
147                 unsuccessfulPrefetches++;
148
149                 delete prefetchBuffer.front();
150                 prefetchBuffer.pop_front();
151             }
152
153             prefetchBuffer.push_back(request);
154             rv = true;
155         }
156         else
157         {
158             if (useTagCache)
159             {
160                 if ((request->flags & NVMainRequest::FLAG_EVICT) &&
                    ↪ !(request->flags & NVMainRequest::FLAG_SKIP_TAG))
161                 {
162                     if (request->type == NVM::WRITE)
163                     {
164                     }
165                     if (request->type == NVM::READ)
166                     {
167                         for (auto it = waitingDRAMWrite.begin(); it !=
                            ↪ waitingDRAMWrite.end(); it++)
168                         {
```

```
169                     if ((*it)->address.GetPhysicalAddress() ==
                        ↪  request->address.GetPhysicalAddress())
170                     {
171                         uint64_t row, col, bank, rank, channel,
                            ↪  subarray;
172
173                         this->IssueCommand(*it);
174
175                         it = waitingDRAMWrite.erase(it);
176
177                         uint64_t targetAddr =
                            ↪  inverseMap[request->address.GetPhysicalAddress()];
178
179                         // Create NVRAM Write request to the appropiate
                            ↪  address
180                         NVMainRequest *NVWrite = new NVMainRequest();
181                         NVWrite->access = UNKNOWN_ACCESS;
182                         GetDecoder()->Translate(targetAddr, &row, &col,
                            ↪  &bank, &rank, &channel, &subarray);
183                         channel = 1; // NVRAM channel
184                         targetAddr = GetDecoder()->ReverseTranslate(
                            ↪  row, col, bank, rank, channel, subarray);
185                         NVWrite->address.SetPhysicalAddress(
                            ↪  targetAddr);
186                         NVWrite->address.SetTranslatedAddress(row, col,
                            ↪  bank, rank, channel, subarray);
187                         NVWtoDRW[targetAddr] =
                            ↪  request->address.GetPhysicalAddress();
188                         NVWrite->status = MEM_REQUEST_INCOMPLETE;
189                         NVWrite->type = NVM::WRITE;
190                         NVWrite->owner = this;
191                         NVWrite->data = request->data; // Read data
                            ↪  from DRAM
192                         NVWrite->flags |= NVMainRequest::FLAG_EVICT;
193                         NVWrite->flags |= NVMainRequest::FLAG_SKIP_TAG;
```

```
194
195                             NVRAMWriteSize++;
196
197                             if (NVRAMWriteSize == 20)
198                                 NVRAMWriteSize--;
199
200                             this->IssueCommand(NVWrite);
201
202                             break;
203                         }
204                     }
205                 }
206             }
207             else
208             {
209             }
210         }
211         delete request;
212         rv = true;
213     }
214 }
215 else
216 {
217     rv = GetParent()->RequestComplete(request);
218 }
```

APÉNDICE F

Código fuente para el fichero «nvmain.h»

```
std::unordered_map<uint64_t,uint64_t> inverseMap;
uint64_t tagLines;

std::vector<NVMainRequest*> waitingDRAMRead;
int DRAMReadSize = 0;
std::vector<NVMainRequest*> waitingDRAMWrite;
int DRAMWriteSize = 0;
std::vector<NVMainRequest*> waitingNVRAMWrite;
int NVRAMWriteSize = 0;
std::vector<NVMainRequest*> declinedReq;
int DeclinedReqSize = 0;
uint64_t totalReqs, reqsHit, reqsMiss, notIssuableTag, tagReplacements;
uint64_t RDHit, RDMiss, WRHit, WRMiss;
uint64_t RD_Replc, RD_Clean, RD_Present;
uint64_t readsOps, readsNotDRAM;

bool useTagCache = false;

uint64_t tmpAddr = 0;

std::unordered_map<uint64_t, uint64_t> NVWtoDRW;
```

```
22
23 class LRU {
24   public:
25     int size;                                    //Size variable
26     std::unordered_map<uint64_t, uint64_t> m; //Data storage
27     std::deque<uint64_t> dq;                   //Implement LRU logic
28
29     LRU(uint64_t capacity) {
30       m.clear();
31       dq.clear();
32       size = capacity;
33     }
34
35     inline bool has(uint64_t key)
36     {
37       return !(m.find(key) == m.end());
38     }
39
40     uint64_t get(uint64_t key, int *found)
41     {
42       if (m.find(key) == m.end()) //Not in cache
43       {
44         *found = 0;
45         return 0;
46       }
47
48       auto it = dq.begin();
49       while(*it != key)
50         it++;
51
52       dq.erase(it);
53       dq.push_front(key);
54
55       *found = 1;
56       return m[key];
```

```
57      }
58
59      void put(uint64_t key, uint64_t value)
60      {
61        if (m.find(key) == m.end()) //Not in cache
62        {
63          if(size == dq.size())     //Cache is full, need to replace
64          {
65            uint64_t leastUsed = dq.back();
66            dq.pop_back();
67            m.erase(leastUsed);
68          }
69        }
70        else                        //Already present, need to delete the
          ↪  already present
71        {
72          auto it = dq.begin();
73          while(*it != key)
74            it++;
75
76          dq.erase(it);
77          m.erase(key);
78        }
79
80        //Add the pair
81        dq.push_front(key);
82        m[key] = value;
83      }
84
85      inline uint64_t removeLRU()
86      {
87        uint64_t leastUsed = dq.back();
88        dq.pop_back();
89        uint64_t freedAddr = m[leastUsed];
90        m.erase(leastUsed);
```

```
91        return freedAddr;
92      }
93
94      inline bool isFull()
95      {
96        return (size == dq.size());
97      }
98  };
99
100 LRU *replacementPolicy;
```