Universidad de Valladolid

Escuela de Ingeniería Informática de Valladolid

Trabajo Fin de Máster

Máster en Ingeniería Informática. Especialidad Big Data

Estudio y Caracterización de Memorias No Volátiles Resistivas con NVMain

Autor:

José Martín Gago

Tutores:

Benjamín Sahelices Fernández

Helena Castan Lanaspa

Agradecimientos

A mi familia por convencerme para seguir estudiando y realizar el máster. A Benjamín y Helena por haberme ayudado durante la realización de este trabajo. A Óscar y Luís por ayudarme en el laboratorio y a Salvador y Hector por su acogimiento en el grupo de investigación.

Resumen

Durante los inicios de la informática el acceso a los datos ha sido el gran cuello de botella a superar, por ello, en un principio surgieron las memorias RAM, pero con el paso del tiempo su arquitectura ha ido evolucionando. Ahora mismo podemos encontrar una nueva generación de memorias que ofrecen un gran número de posibilidades con respecto a las anteriores, las RRAM o ReRAM (memorias resistivas), memorias no volátiles que no pierden su estado tras una desconexión completa del equipo. En este documento se presentará en primer lugar a modo introductorio una evolución de las memorias, en segundo lugar la documentación correspondiente al simulador NVMain, del cual se ha realizado un estudio completo con respecto a los simuladores de memoria que provee, y por último las modificaciones que se realizan sobre el controlador de memoria y las pruebas ejecutadas. Junto con las partes descritas anteriormente, se incluye la planificación, riesgos y demás elementos correspondientes a un documento de tésis. Este trabajo se centra principalmente en el controlador de memoria y la gestión que realiza para lanzar las transacciones.

Abstract

Along the begining of the informatic the access to the data has been the big bottle-neck to overcome, for that reason, the RAM memory appear to minimize this problem but thier architecture has been changing all this time tom improve this data access. Now we can find a new generation of memories that offer a big number of possibilities against the old ones, the RRAM or ReRAM (resistive memories), this kind of memories don't lose their state after a disconnection of the host. For this reason this memories are non volatile memories. First of all, in this document will be presented the evolution of the memories, secondly, the documentation of the NVMain simulator, which has been completed a full study of the memory controlers that are provided, and the last topics will be the modifications on this memory controlers and the executed tests. Besides the parts described avobe, the planification, the risks and other elements corresponding to a thesis document are included. This thesis is focused in the memory controler and the management that makes to launch the transactions.

Índice general

Índice de figuras

Índice de cuadros

Parte I

Introducción y Planificación

Capítulo 1

Introducción

A lo largo del tiempo, tanto los elementos de computación como los elementos de almacenamiento de datos han ido evolucionando alcanzando mejores rendimientos. Esta evolución ha sido mayor en la computación provocando que el cuello de botella se produzca en el acceso a datos, por lo tanto los estudios en esta área de la informática son importantes para posibilitar encontrar nuevos modelos y arquitecturas de almacenamiento que aporten nuevos beneficios.

Para ello se empleará un simulador de memoria principal llamado NVMain y para el estudio que se realizará en este documento se emplearán memorias RAM no volátiles (RRAM o ReRAM). Actualmente, este tipo de memoria principal es una memoria volátil donde se almacenan tanto los datos que están manejando los programas que se encuentran en la CPU (Unidad de Procesamiento Central) como los datos de aquellos programas que se van a procesar. Al emplear este tipo de memorias para el propósito descrito, aparece una de las principales características de las memorias RRAM, los datos dejan de ser volátiles, no se pierden si hay un corte del suministro eléctrico y se mantienen en esta memoria hasta que se sobrescriban.

NVMain está implementado utilizando el lenguaje de programación orientado a objetos *C++*. Tiene una estructura bien diferenciada con una jerarquía en forma de árbol y sus componentes pueden ser reemplazados de manera sencilla.

1.1. Contexto

Aunque encontramos una gran variedad de simuladores de memoria, el gran problema en este sentido es su diseño, pues no están pensados desde la base para este tipo de memorias no volátiles sino que reutilizan simuladores de memorias DRAM, por ello no son eficaces para realizar simulaciones con memorias RRAM, mientras que NVMain sí está pensado para ello, además permite una gran flexibilidad posibilitando la implementación de diferentes controladores de memoria, interconnects y más componentes del simulador que se verán más adelante o también es posible crear nuestros propios objetos sustituyendo los que el propio simulador provee de manera sencilla.

1.2. Motivación

La principal motivación para el desarrollo de este Trabajo de Fin de Máster es la profundización en el entendimiento del funcionamiento de las memorias, centrándose de manera concreta en las memorias no volátiles y en el simulador de memoria NVMain. Se pretende que el estudio de este simulador ayude a comprender de mejor manera el funcionamiento de los controladores de memoria los procesos que se llevan a cabo por estos y como se emulan para poder simular sus comportamientos, las diferentes políticas que emplean sobre las peticiones y como se podrían diseñar nuevos modelos de controladores.

Se espera que al comprender el funcionamiento de las memorias y del simulador se consigan obtener conocimientos sobre este nexo entre los componentes hardware y el diseño software para controlarlo, como interaccionan y como modelar estos sistemas para poder llevar a cabo tareas de emulación, previsión de su funcionamiento y rendimientos esperados. Una tarea que resulta muy importante cuando se va a lanzar un nuevo producto, pues este debe ser ampliamente estudiado para asegurar su éxito.

1.3. Objetivos

Los objetivos son:

- Llevar a cabo un estudio completo de este simulador comprendiendo su arquitectura, su funcionamiento, sus diferentes componentes, como interactúan estos componentes, los diferentes parámetros del archivo de configuración, los archivos de entrada al simulador y como llevar a cabo modificaciones en el mismo.
- Entender el funcionamiento de las memorias resistivas (RRAM), conocer sus diferencias con respecto a las memorias actuales y cuales son sus ventajas e inconvenientes.
- Realizar simulaciones para comprobar y entender las diferencias de rendimiento de los controladores de memoria.

Capítulo 2

Planificación del proyecto

2.1. Propuesta inicial

Para el desarrollo de este proyecto se ha empleado un modelo de planificación ágil, teniendo reuniones semanales con el tutor y estableciendo hitos en las mismas para finalizar en el tiempo antes de la siguiente reunión.

2.2. Desarrollo real

No fue posible la realización del proyecto siguiendo completamente la planificación que se estableció en un principio, debido a la complejidad de algunas tareas y a tener una fecha límite para la entrega y presentación del mismo. Por lo tanto, se optó por reducir en cierta medida la carga de trabajo referente a la modificación de los controladores de memoria.

El desarrollo del proyecto se inició a mediados de febrero de 2017, desde entonces se llevó a cabo el estudio de las memorias en primer lugar hasta mediados de marzo y tras esto el estudio del simulador de memoria NVMain, el cual se extendió más de lo esperado debido al gran número de clases con las que cuenta y la complejidad del mismo en ciertos aspectos. A finales de Junio se finalizó con el estudio del simulador, por lo que se inició la creación del script en Python para la comparación de los diferentes controladores de memoria empleando un mismo archivo de configuración y diferentes políticas.

A mediados de julio se terminaron de manera definitiva los nuevos objetivos del proyecto junto con el documento.

2.3. Artefactos

Los artefactos propuestos en un principio son los siguientes:

- Instalación de NVMain y GEM5.
- Estudio de los diferentes tipos de memorias.
- Estudio inicial de componentes de NVMain.

- Estudio de los controladores de memoria.
- Estudio de la decodificación de direcciones.
- Simulaciones de prueba.
- Desarrollo y empleo del script para comparar los diferentes controladores de memoria.
- (*) Creación de un nuevo controlador de memoria a partir de uno existente añadiendo modificaciones al mismo.

El último artefacto no pudo ser realizado debido a la falta de tiempo como ya se indicó anteriormente.

2.4. Tareas

Como se podrá observar más adelante, se han producido retrasos en determinadas etapas del proyecto. La causa de estos retrasos han sido causados por el impacto de los riesgos sobre la planificación que se realizó en un primer momento. Lo que se ha traducido en:

- Retraso de la fase de planificación y preliminares.
- Retraso de la fase de análisis.
- Retraso y eliminación de objetivos de la fase de implementación.

Los planes de contingencia para los riesgos acontecidos se pueden encontrar en la siguiente sección 2.6.

2.4.1. Actividades periódicas

Un día a la semana se ha llevado a cabo una reunión con el tutor Benjamín Sahelices para establecer objetivos y resolver dudas. Además, también se ha realizado una reunión a la semana con el resto de miembros del grupo de caracterización de materiales y dispositivos electrónicos, entre los que se encuentra la tutora Helena Castan, para poner en común los distintos avances.

2.4.2. Desglose de tareas del proyecto

En esta parte del documento se muestra en detalle la planificación realizada, tanto la planificación inicial como la planificación real del proyecto para las distintas etapas de este. Para poder establecer esta planificación se ha empleado Microsoft Project 2016.

Planificación inicial

En la figura 2.1 se puede observar las diferentes fases principales con sus periodos de duración y los periodos de tiempo establecidos. La figura 2.2 muestra en detalle la fase de planificación

y preliminares junto con las subtareas que la forman. La figura 2.3 contiene las subtareas relacionadas con la fase de análisis. En la figura 2.4 referencia a la fase de implementación y sus subtareas. La fase de pruebas corresponde a la figura 2.5 donde encontramos las subtareas que la forman. El diagrama de Gant correspondiente a esta planificación inicial puede observarse en la figura A.1.

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		**Reuniones con el tutor**	**106 días**	**vie 17/02/17**	**vie 14/07/17**	
24		**Reuniones con el grupo**	**101 días**	**mié 22/02/17**	**mié 12/07/17**	
46		**Fase de planificación y preliminares**	**18 días**	**vie 17/02/17**	**mar 14/03/17**	
52		**Fase de análisis**	**59 días**	**mié 15/03/17**	**lun 05/06/17**	**46**
58		**Fase de implementación**	**29 días**	**mar 06/06/17**	**vie 14/07/17**	**52**
62		Elaboración del documento académico	106 días	vie 17/02/17	vie 14/07/17	
63		**Fase de pruebas**	**25 días**	**lun 12/06/17**	**vie 14/07/17**	

Figura 2.1: Vista global de las tareas del proyecto

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
46		**Fase de planificación y preliminares**	**18 días**	**vie 17/02/17**	**mar 14/03/17**	
47		Planificación	4 días	vie 17/02/17	mié 22/02/17	
48		Estudio de memorias SDRAM	6 días	jue 23/02/17	jue 02/03/17	
49		Estudido de memorias RRAM	6 días	vie 03/03/17	vie 10/03/17	
50		Estudio de conceptos	1 día	lun 13/03/17	lun 13/03/17	
51		Fin de planificación y preliminares	0 días	mar 14/03/17	mar 14/03/17	48;49;50

Figura 2.2: Fase de planificación y preliminares

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
52		**Fase de análisis**	**59 días**	**mié 15/03/17**	**lun 05/06/17**	**46**
53		Instalación de NVMain y GEM5	1 día	mié 15/03/17	mié 15/03/17	
54		Estudio inicial de NVMain	17 días	jue 16/03/17	vie 07/04/17	
55		Estudio de los controladores de memoria	25 días	lun 10/04/17	vie 12/05/17	
56		Estudio del decodificador de direcciones	6 días	vie 26/05/17	vie 02/06/17	
57		Fin de análisis	0 días	lun 05/06/17	lun 05/06/17	53;54;55;56

Figura 2.3: Fase de análisis

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
58		**Fase de implementación**	**29 días**	**mar 06/06/17**	**vie 14/07/17**	**52**
59		Desarrollo del script	7 días	mar 06/06/17	mié 14/06/17	
60		Creación de nuevo controlador de memoria	21 días	jue 15/06/17	jue 13/07/17	
61		Fin fase de implementación	0 días	vie 14/07/17	vie 14/07/17	59;60

Figura 2.4: Fase de implementación

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
63		**Fase de pruebas**	**25 días**	**lun 12/06/17**	**vie 14/07/17**	
64		Bateria de pruebas script	3 días	lun 12/06/17	mié 14/06/17	
65		Bateria de pruebas controlador de memoria	5 días	vie 07/07/17	jue 13/07/17	
66		Fin fase de pruebas	0 días	vie 14/07/17	vie 14/07/17	64;65

Figura 2.5: Fase de pruebas

Planificación real

A continuación se muestra la planificación real resultante debido a como afectaron los riegos al desarrollo. La figura 2.6 muestra la nueva disposición temporal de las fases del proyecto. La figura 2.7 muestra en detalle la fase de planificación y preliminares junto con las subtareas que la forman y las nuevas fechas de inicio y fin. La figura 2.8 contiene las subtareas relacionadas con la fase de análisis y los nuevos periodos de tiempo para esta fase debidos a los riesgos. En la figura 2.9 referencia a la fase de implementación y sus subtareas, los artefactos correspondientes a esta fase han tenido que ser recortados por imposibilidad temporal debido a los riegos que han afectado al proyecto. La fase de pruebas corresponde a la figura 2.10 donde encontramos las subtareas que la forman. El diagrama de Gant resultante de esta planificación se puede ver en la figura A.2.

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		**Reuniones con el tutor**	**106 días**	**vie 17/02/17**	**vie 14/07/17**	
24		**Reuniones con el grupo**	**101 días**	**mié 22/02/17**	**mié 12/07/17**	
46		**Fase de planificación y preliminares**	**24 días**	**vie 17/02/17**	**mié 22/03/17**	
52		**Fase de análisis**	**70 días**	**jue 23/03/17**	**mié 28/06/17**	**46**
58		**Fase de implementación**	**12 días**	**jue 29/06/17**	**vie 14/07/17**	**52**
61		Elaboración del documento académico	106 días	vie 17/02/17	vie 14/07/17	
62		**Fase de pruebas**	**6 días**	**vie 07/07/17**	**vie 14/07/17**	

Figura 2.6: Vista global de las tareas del proyecto

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
46		**Fase de planificación y preliminares**	**24 días**	**vie 17/02/17**	**mié 22/03/17**	
47		Planificación	5 días	vie 17/02/17	jue 23/02/17	
48		Estudio de memorias SDRAM	8 días	vie 24/02/17	mar 07/03/17	
49		Estudido de memorias RRAM	9 días	mié 08/03/17	lun 20/03/17	
50		Estudio de conceptos	1 día	mar 21/03/17	mar 21/03/17	
51		Fin de planificación y preliminares	0 días	mié 22/03/17	mié 22/03/17	48;49;50

Figura 2.7: Fase de planificación y preliminares

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
52		**Fase de análisis**	**70 días**	**jue 23/03/17**	**mié 28/06/17**	**46**
53		Instalación de NVMain y GEM5	1 día	jue 23/03/17	jue 23/03/17	
54		Estudio inicial de NVMain	24 días	vie 24/03/17	mié 26/04/17	
55		Estudio de los controladores de memoria	33 días	jue 27/04/17	lun 12/06/17	
56		Estudio del decodificador de direcciones	11 días	mar 13/06/17	mar 27/06/17	
57		Fin de análisis	0 días	mié 28/06/17	mié 28/06/17	53;54;55;56

Figura 2.8: Fase de análisis

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
58		**Fase de implementación**	**12 días**	**jue 29/06/17**	**vie 14/07/17**	**52**
59		Desarrollo del script	10 días	jue 29/06/17	mié 12/07/17	
60		Fin fase de implementación	0 días	vie 14/07/17	vie 14/07/17	59

Figura 2.9: Fase de implementación

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
63		**Fase de pruebas**	**25 días**	**lun 12/06/17**	**vie 14/07/17**	
64		Bateria de pruebas script	3 días	lun 12/06/17	mié 14/06/17	
65		Bateria de pruebas controlador de memoria	5 días	vie 07/07/17	jue 13/07/17	
66		Fin fase de pruebas	0 días	vie 14/07/17	vie 14/07/17	64;65

Figura 2.10: Fase de pruebas

2.5. Gestión de recursos

En esta sección del documento se indicarán loas horas empleadas para el desarrollo del proyecto junto con la amortización de los equipos de trabajo.

2.5.1. Análisis de costes

Los costes asociados al proyecto se desglosan en las horas de trabajo y la amortización de los equipos empleados para la finalización del proyecto. Tiendo en cuenta un salario de 35 euros la hora, el coste de personal en función de las horas de trabajo establecidas en el siguiente subapartado (Horas de trabajo) hace un total de 26.740 euros.

Horas de trabajo

Podemos dividir las horas de trabajo entre presenciales y no presenciales. El número de horas realizadas se estima en 550 horas presenciales y 214 horas no presenciales, lo que hace un total de 764 horas. El horario presencial en el laboratorio ha sido de 09:00 a 14:00 durante todo el desarrollo y además, a partir de mediados de junio hasta finales del mismo mes se incluye en este horario el intervalo de 16:00 a 18:00 en el laboratorio.

Amortización de equipos

Los equipos que se han empleado son el equipo del laboratorio desde el que se ha trabajado y la máquina Zujar en la que se han realizado las simulaciones. Ha sido muy importante la disponibilidad de esta máquina en concreto, ya que se trata de la única máquina donde se encuentra el simulador.

2.6. Gestión de riesgos

En esta sección se detallarán los riesgos potenciales que afectan al proyecto, estos riesgos se dividen en diferentes tipos, pueden tratarse de riesgos de planificación, del personal, de proceso, externos y del material de trabajo. El método de calcular la magnitud de estos riesgos ha sido empleando la figura 2.11.

		Impacto				
		Muy bajo	Bajo	Medio	Alto	Muy alto
Probabilidad	Muy baja	Baja	Baja	Media	Media	Media
	baja	Baja	Baja	Media	Media	Alata
	Media	Baja	Media	Media	Alta	Alta
	Alta	Media	Media	Alta	Alta	Muy alta
	Muy alta	Media	Media	Alta	Muy alta	Muy alta

Figura 2.11: Magnitud de los riesgos.

2.6.1. Riesgos identificados

Riesgos de planificación

R-01. Imposibilidad de realizar los hitos en el tiempo planificado

Fase: Elaboración y construcción.

Probabilidad: Alta.

Magnitud: Alta.

Descripción: Durante la realización de los hitos estos resultan ser demasiado complejos, por lo que llevan más tiempo del esperado.

Impacto: El impacto sobre el proyecto es alto debido a que retrasaría la fecha de finalización la cual ya está ajustada.

Indicadores: La duración de la tarea se retrasa demasiado provocando que el inicio de las siguientes se retrase también.

Plan de mitigación: Llevar a cabo un seguimiento de las tareas a realizar para evitar salirse de plazo.

Plan de contingencia: Replanificar las tareas para poder terminar en la fecha establecida.

R-02. Imposibilidad de llevar a cabo todos los hitos en el tiempo planificado

Fase: Elaboración y construcción.

Probabilidad: Alta.

Magnitud: Muy alta.

Descripción: Durante la realización del proyecto no es posible terminar realizando todas las tareas para finalizar todos los hitos.

Impacto: El impacto es muy alto ya que significa la imposibilidad de terminar las tareas planificadas.

Indicadores: La fecha de finalización está próxima y las tareas a realizar están retrasadas.

Plan de mitigación: Llevar a cabo las tareas en función del orden de importancia siempre que sea posible porque no sea necesaria la tarea anterior.

Plan de contingencia: Mejorar las tareas ya realizadas obviando las no completadas.

Riesgos del personal

R-03. Imposibilidad de reunirse

Fase: Todas.

Probabilidad: Baja.

Magnitud: Baja.

Descripción: Debido a la no disponibilidad de los miembros del grupo de investigación o del tutor no es posible llevar a cabo la reunión semanal.

Impacto: El impacto que se produce es medio, ya que se puede continuar con el resto de tareas restantes en caso de haber finalizado con una.

Indicadores: Alguna persona no puede reunirse esa semana.

Plan de mitigación: Establecer un plan de trabajo para su desarrollo en más de una semana.

Plan de contingencia: Tener otra reunión cuando sea posible.

R-04. Baja por enfermedad

Fase: Todas

Probabilidad: Baja.

Magnitud: Media.

Descripción: Es necesario un periodo de baja debido a una enfermedad.

Impacto: El impacto es alto, debido a que se pierden varios días de trabajo.

Indicadores: Algún miembro comunica que no se encuentra bien y no puede acudir.

Plan de mitigación: Finalizar las tareas en las fechas indicadas para poder realizar una replanificación si es necesario y establecer un plan de trabajo de desarrollo de más de una semana.

Plan de contingencia: Dependiendo del miembro del equipo que se ponga enfermo:

- El estudiante: replanificar las tareas para terminar en la fecha establecida.
- El tutor: consultar a la otra tutora del proyecto si hay alguna duda.

R-05. Carencia de conocimientos para realizar una tarea.

Fase: Todas.

Probabilidad: Media.

Magnitud: Alta.

Descripción: La tarea es demasiado compleja para el estudiante, por lo que resulta más compleja de realizar.

Impacto: El impacto es alto debido a que provoca que una tarea se alargue más en el tiempo.

Indicadores: La fecha de finalización está próxima y las tareas a realizar están retrasadas.

Plan de mitigación: A la hora de realizar la planificación tener en cuenta si se conoce lo que se debe realizar o no para llevar a cabo una planificación más adecuada.

Plan de contingencia: En primer lugar consultar con el tutor para evitar este riesgo, si no es posible evitar el riesgo se replanificará el proyecto.

Riesgos de proceso

R-06. Retrasos en la fase de elaboración

Fase: Elaboración.

Probabilidad: Media.

Magnitud: Alta.

Descripción: Surgen problemas que provocan el retraso de la fase de elaboración.

Impacto: El impacto es muy alto, ya que retrasa todas las tareas de elaboración restantes y de diseño.

Indicadores: Las tareas de elaboración se retrasan.

Plan de mitigación: Realizar una buena planificación de los hitos a realizar.

Plan de contingencia: Suprimir alguno de los artefactos a realizar.

Riesgos externos

R-07. Terremotos, inundaciones o incendios

Fase: Todas.

Probabilidad: Muy baja.

Magnitud: Media.

Descripción: Los equipos de trabajo son destruidos debido a alguno de estos fenómenos.

Impacto: El impacto es muy alto, puesto que supondría un retraso debido a la perdida del trabajo realizado y la imposibilidad temporal de continuar con el proyecto.

Indicadores: No se identifican indicadores para estos sucesos.

Plan de mitigación: Realizar una copia periódica del trabajo para evitar perder el trabajo realizado.

Plan de contingencia: Recuperar los datos disponibles en la copia de seguridad.

Riesgos del material de trabajo

R-08. Corrupción del sistema de ficheros de la máquina Zujar.

Fase: Elaboración y construcción.

Probabilidad: Baja.

Magnitud: Media.

Descripción: Algún error software provoca la pérdida de la máquina Zujar.

Impacto: El impacto es alto ya que sería necesario llevar a cabo la reinstalación de los paquetes.

Indicadores: No es posible realizar el arranque o la máquina se queda congelada y no responde.

Plan de mitigación: Tener bien documentada la reinstalación de los paquetes para poder reinstalar todo de manera rápida.

Plan de contingencia: Solicitar a los técnicos una nueva instalación limpia de Zujar para instalar los paquetes necesarios nuevamente.

R-09. Fallo hardware de las máquinas de trabajo.

Fase: Todas.

Probabilidad: Baja.

Magnitud: Baja.

Descripción: La máquina de trabajo del laboratorio o el portátil empleado sufren alguna avería.

Impacto: El impacto es bajo ya que se dispone de una máquina de respaldo.

Indicadores: Alguna de estas máquinas no responde.

Plan de mitigación: Tener otra máquina de respaldo para poder continuar con el trabajo.

Plan de contingencia: Utilizar esta segunda máquina para continuar con las tareas.

Parte II

Preliminares

Capítulo 3

Memorias SDRAM

3.1. Conceptos

En el siguiente apartado se procederá a describir el funcionamiento de las memorias SDRAM, pero para ello es necesario comprender antes otros tipos de memorias [18] y otros conceptos importantes como los sense amplifiers [19].

3.1.1. Sense amplifiers

En este documento se va a referenciar a estos componentes en varias ocasiones, es importante comprender que son estos elementos y cual es su función.

Un sense amplifier es uno de los elementos que componen la circuitería del chip de memoria del semiconductor, es un circuito integrado. Estos sense amplifiers son parte del circuito de lectura que es usado por ejemplo cuando se leen datos de la memoria, el rol de estos elementos es detectar las señales de energía de baja potencia de una línea de bits que representa un bit de datos almacenado en una celda de memoria y amplificar el pequeño voltaje a niveles lógicos reconocibles para poder interpretar correctamente los datos.

Los sense amplifiers modernos consisten de entre dos a seis transistores, mientras que los sense amplifiers para la memoria del núcleo pueden contener hasta 13 transistores. Hay un sense amplifier para cada columna de celdas de memoria, l oque significa que hay cientos o miles de sense amplifiers idénticos en un chip de memoria moderno.

Estos sense amplifiers son el nexo que permite acceder a las celdas de memoria, ya sea para leer sus datos o para almacenar otros datos. Por lo tanto, para llevar a cabo una operación de lectura es necesario esperar a que los datos se carguen en el sense amplifier para poder leer y para realizar una operación de escritura es necesario esperar a que los sense amplifiers almacenen los datos en las celdas de memoria antes de realizar otra operación.

3.1.2. Diferentes arquitecturas de memorias

Memorias RAM

En primer lugar es necesario entender qué es una memoria RAM (Random Access Memory), estas memorias son una manera de almacenamiento de datos en el ordenador. Se trata de circuitos integrados que permiten ser accedidos en cualquier orden, *Random* hace referencia a la idea por la cual cualquier porción de datos puede ser devuelta en un tiempo constante, sin importar su localización física o si está o no relacionado con la porción de datos anterior.

Memorias SRAM

Las memorias SRAM (Static RAM), trabajan bajo el principio de un conmutador que se enciende o apaga y que requiere de 2 a 4 transistores por bit (un bit tiene dos estados, encendido o apagado, se trata de la unidad de memoria más pequeña y está formado por una célula de memoria).

Memorias DRAM

Las memorias DRAM (Dynamic RAM) de manera distinta a las SRAM, se basan en la habilidad de los condensadores para mantener una carga y solamente requieren un transistor por bit. Como las células de memoria DRAM son más pequeñas que las células de memoria SRAM los fabricantes pueden colocar un mayor número de células de memoria en el mismo espacio, lo que se traduce en una mayor capacidad y un menor coste por bit. Sin embargo, los condensadores pierden carga periódicamente, por lo tanto es necesario llevar a cabo un refresco de esta carga cada cierto tiempo. Debido a este refresco, se trata de memorias dinámicas. Un símil para entender este fenómeno sería un cubo de agua con agujeros en el fondo [18], este cubo puede estar vacío (0) o lleno (1), pero cuando esté lleno irá perdiendo agua poco a poco, por lo que será necesario añadir más agua de vez en cuando antes de que se agote para mantener el estado de cubo con agua (1). Si este proceso de añadir agua (refresco de memoria) no se realiza de una manera lo suficientemente frecuente, el cubo perderá todo el agua (pasará a 0) y no será posible saber que el estado correcto es lleno (1). En DRAM se utiliza este refresco para mantener el estado de la carga y con ello la información almacenada, mientras que con SRAM no es necesario llevar a cabo este proceso pues no se produce esta pérdida de carga al usar diferentes componentes electrónicos. Este refresco se traduce en un coste constante de energía para mantener el estado de los bits. Además, esta pérdida de energía provoca que a la hora de realizar lecturas, sea necesario interpretar el estado, por lo que hace falta el empleo de detectores de nivel, si el condensador tiene la mitad o menos de la energía, se considera un 0, en caso contrario es un 1. Lo cual se traduce en un incremento de los tiempos de lectura, una de las razones por la que SRAM es más rápido. A continuación podemos ver una unidad básica de memoria DRAM en la figura 3.1 [18].

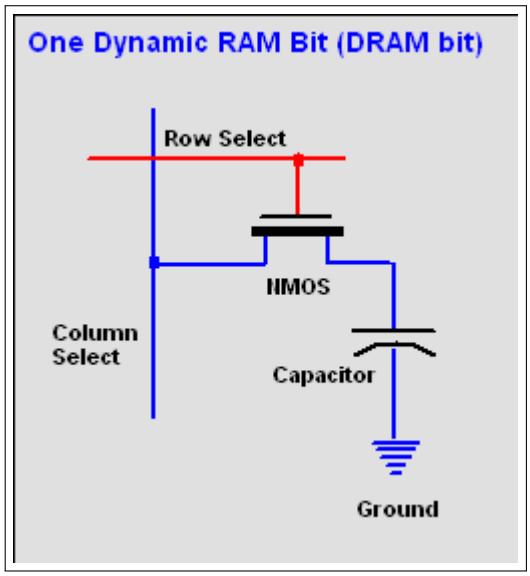

Figura 3.1: Unidad básica de DRAM

3.1.3. Memorias SDRAM

Las memorias SDRAM (Synchronous DRAM) son una DRAM que trabaja de manera síncrona con el bus. Las memoria DRAM tienen una interfaz asíncrona, esto significa que responde tan rápido como es posible para para cambiar las entradas de control. La interfaz síncrona de estas memorias provoca que esperen hasta una señal de reloj antes de responder a las entradas de control y después se sincronizan con el bus. El reloj es usado para correr un estado de máquina finito interno que encola las instrucciones entrantes, lo que permite que el chip tenga un patrón de complejidad mayor de operación que la DRAM asíncrona, esto se traduce en un mayor ancho de banda de estas memorias debido a la posibilidad de enviar una mayor ráfaga de datos tras la sincronización, esta es la causa de conseguir mayores velocidades. El uso de colas habilita que el chip pueda aceptar una nueva instrucción antes de la finalización del procesamiento de la anterior. El uso de colas para escritura permite que un comando de escritura sea seguido por otra instrucción sin esperar a que acaben de escribirse los datos en el array de memoria, y en una cola de lectura los datos solicitados aparecen después de un número fijo de pulsos de reloj tras la instrucción de lectura, durante esos ciclos de espera de los datos se pueden enviar más instrucciones. Este retraso de espera por los datos se conoce como latencia.

Datos técnicos de SDRAM

Como ya se indicó anteriormente las memorias SDRAM son una variación de las DRAM, utilizan un transistor y un condensador para almacenar los bits de datos, en lugar de los seis que emplean las SRAM. Dada una dirección de memoria en la cual leer o escribir, el transistor se abre y el el condensador es cargado con el valor de entrada, o el voltaje almacenado es recogido. Para mitigar el efecto de pérdida energética de los condensadores, todas las filas son refrescadas ocasionalmente manteniendo el voltaje de estos aunque no haya peticiones externas. Otra de

las capacidades de las SDRAM es la habilidad para encolar comandos en lugar de manejar una petición en cada momento. Aunque la propia SDRAM no lleva a cabo este manejo a alto nivel de las peticiones externas, esta gestión de las SDRAM y la garantía de que las peticiones se manejan de manera adecuada se realiza por un controlador, el cual traduce las peticiones del host en señales de control para la SDRAM mientras mantiene un seguimiento del estado de la memoria. Este controlador también lleva a cabo el proceso inverso, traduciendo las señales de la SDRAM al host. Por todo esto, no se puede realizar un análisis de las memorias SDRAM sin analizar también el controlador.

Señales de control de la SDRAM

Antes de llevar a cabo el análisis del controlador de memoria, se describirán las señales de control que se emplean en la comunicación entre el host y el controlador:

- *CKE*. Reloj activado, esta señal enmascara la señal de reloj de la SDRAM, mientras está a 0 no se pueden detectar flancos de subida de reloj y la SDRAM se detiene hasta que vuelve a ser 1.
- *CS*, Chip seleccionado, cuando la señal está a 1, la SDRAM ignora todas las señales recibidas desde el controlador excepto *CKE*.
- *DQM*. Máscara de datos, cuando está a 1 se suprimen las lecturas/escrituras de datos, los comandos de escritura no se ejecutan y si se ocurren dos ciclos antes de un ciclo de lectura no se lee ningún dato.
- *RAS* y *CAS*. Dirección de columna/fila estroboscópica. Junto con *WE* sirve para determinar qué comando recibe la SDRAM desde el controlador.
- *WE*. Escritura activada.

Además de las señales de control, la SDRAM también recibe a través del bus la fila, la columna y la dirección del bank sobre los que se va a realizar la operación desde el controlador y los datos que se van a escribir. La columna de direcciones está compuesta por 11 bits y por lo tanto puede ser representada empleando A0-A9 y A11, es importante tener en cuenta que algunos comandos no necesitan una dirección. En la figura 3.2 [18] se muestran los comandos que se pueden enviar desde el controlador a la memoria SDRAM.

Los comandos que se envían son los siguientes:

- *ACTIVATE*. Este comando indica a la SDRAM la dirección de un bank y de una fila, el contenido de la fila del bank de la memoria es leído en el array de la columna contenido en los sense amplifiers del bank, con lo que los datos son accesibles. Como efecto secundario, esta activación provoca un refresco de la carga en la fila de transistores ayudando a prevenir la pérdida de datos. La activación no se produce de manera inmediata, por lo que durante el periodo de tiempo en el cual se está produciendo este paso de datos del array a la columna las operaciones de lectura y escritura no se pueden realizar sobre el bank, aunque los otros

cs	ras	cas	we	A10	Address bits	Bank bits	command
'1'	X	X	X	X	X	X	No operation
'0'	'1'	'1'	'1'	X	X	X	No operation
'0'	'1'	'1'	'0'	X	X	X	Burst terminate
'0'	'1'	'0'	'1'	'0'	Column	Bank address	Read burst
'0'	'1'	'0'	'1'	'1'	Column	Bank address	Read burst with precharge
'0'	'1'	'0'	'0'	'0'	Column	Bank address	Write burst
'0'	'1'	'0'	'0'	'1'	Column	Bank address	Write burst with precharge
'0'	'0'	'1'	'1'	Row		Bank address	Activate row
'0'	'0'	'1'	'0'	'0'	X	Bank address	Precharge selected bank's row
'0'	'0'	'1'	'0'	'1'	X	X	Prechage all bank's row
'0'	'0'	'0'	'1'	X	X	X	Auto refresh each bank's row
'0'	'0'	'0'	'0'	Mode		0,0	Load setting to registers A0-A9

Figura 3.2: Unidad básica de DRAM

banks si son accesibles. La imposición de esta limitación temporal descrita anteriormente en las peticiones de llegada se gestiona a través del controlador de la SDRAM.

- *READ/WRITE*. Cuando un bank ha sido activado, las peticiones a la fila pueden ser manejadas. Un petición de lectura provocará que la SDRAM envíe los datos en la fila a las líneas de datos (líneas *DQ*) tras un tiempo de latencia (*CAS*, 2 o 3 ciclos de reloj). Una petición de escritura provoca que los datos en las líneas de datos sean escritos en la fila del bank.

- *Precharging*. Después de ser activada, una fila debe ser cerrada antes de que se pueda activar una nueva fila. Esto es debido a que los sense amplifiers en el bank tienen que volver al estado libre para que el siguiente contenido de fila pueda ser leído. Hay dos maneras por las cuales la SDRAM realice esta precarga:

  - Mediante el comando de precarga.
  - Añadiendo una petición de lectura/escritura junto con la petición de precarga.

- *Auto refresh*. Realiza un refresco de una fila en cada bank, es necesario que los banks deban ser precargados antes de poder realizar el refresco. La SDRAM posee un contador interno mediante el cual provoca que se cambie la fila seleccionada en cada intervalo de refresco y por ello, el controlador consigue prevenir la pérdida de datos que se produce por la pérdida de voltaje de los condensadores mediante el envío de un refresco automático en cada intervalo de refresco, de esta manera las filas son refrescadas tan pronto como es posible.

- *Model loading*. La SDRAM posee 10 bits (M0-M9) que definen el modo en el que están trabajando.

- M0-M2, indica el tamaño de burst de lectura y de escritura. Si M9 es 1, el tamaño puede ser de 1, 2, 4 u 8 palabras.
- M3, indica si la SDRAM emplea burst secuencial o burst intercalada.
- M4-M6, indica la latencia *CAS*.
- M7 y M8, bit reservado y debe ser 0 cuando el modo ha sido cargado.
- M9, indica si la SDRAM estará en modo burst de escritura o no (la lectura siempre está en burst).

Controlador de memoria

El controlador es el nexo entre la memoria SDRAM y el usuario (host) que envía peticiones de lectura/escritura desde la SDRAM. Generalmente es implementado en el código como una máquina con estado. Este controlador debe mantener un rastreo de la SDRAM para manejar las peticiones del host, lo que incluye:

1. Fila y bank actualmente activos en la SDRAM.
2. Cualquier operación no finalizada en la SDRAM y los ciclos restantes antes de completarse.
3. El estado actual de la SDRAM (inicialización, refresco, etc.).

Todas las peticiones en el host son recibidas en el controlador, este determina si la SDRAM puede manejar las peticiones y actuar en consecuencia. El controlador traduce cada petición en comandos que permiten a la SDRAM gestionar estas peticiones y enviar las señales de control apropiadas, esperando entre los diferentes comandos la cantidad adecuada de ciclos si es necesario. En la figura 3.3 [18] podemos observar un diagrama de una memoria SDRAM junto con su controlador de memoria (ejemplo didáctico) y la interfaz de comunicación con el host.

Operación de escritura

A continuación se describe un ejemplo de una petición de escritura, en el cual la fila y el bank solicitados ya han sido activados en la SDRAM y esta no está manejando un comando anterior. El proceso es el siguiente:

- Primer ciclo: El host envía al controlador una señal *WR* junto con la dirección de escritura, los datos en el bus *hddr* y los datos a ser escritos en el bus *hdin*.
- Segundo ciclo: El controlador envía a la SDRAM la dirección a la que escribir en el bus *Saddr*, los datos a escribir en el bus *Sdata* y los valores apropiados en la señal de control para empezar una operación de escritura.
- Tercer a quinto ciclo: La SDRAM recibe el comando de escritura y escribe los datos.

En la figura 3.4 [18] se muestra el proceso que se acaba de describir.

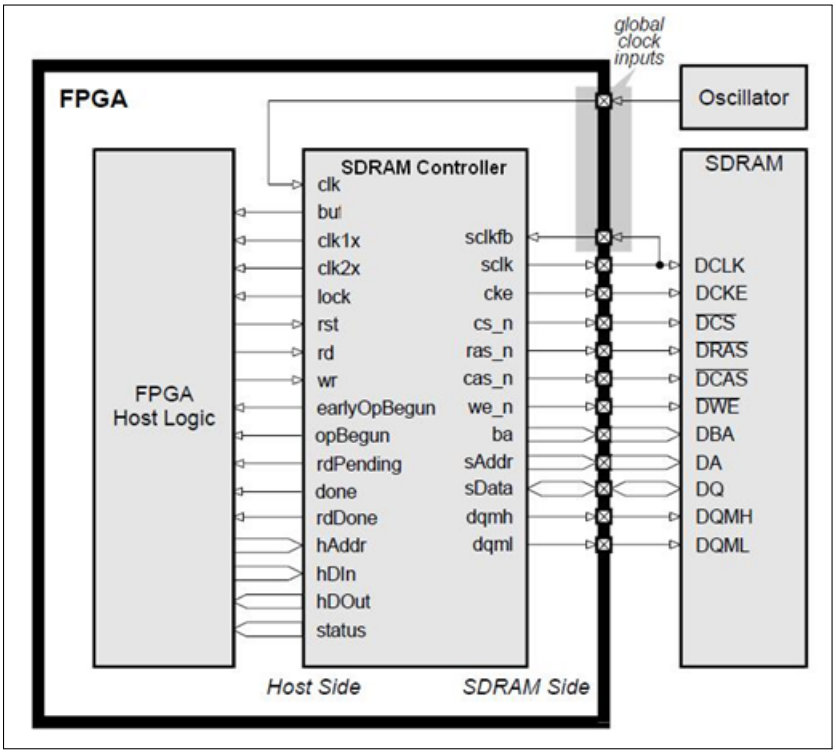

Figura 3.3: Ejemplo didáctico de controlador de memoria SDRAM

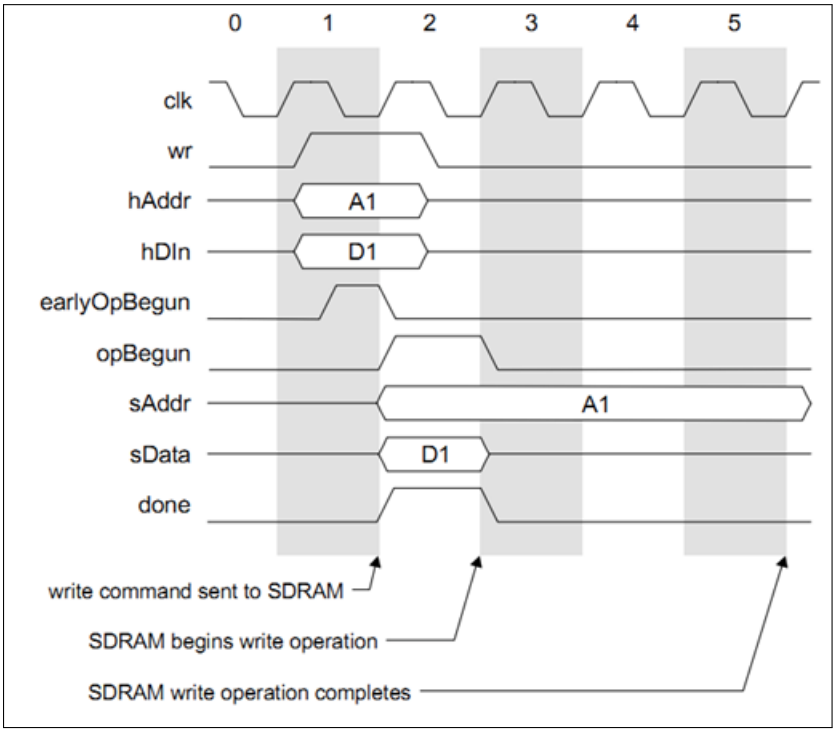

Figura 3.4: Operación de escritura

Capítulo 4

Memorias Resistivas

4.1. Conceptos básicos

Las memorias resistivas de acceso aleatorio son un tipo de memorias no volátiles (ReRAM o RRAM) [17] [24]. Su funcionamiento consiste en el cambio de la resistencia a través de un dieléctrico de estado sólido, también conocido como memristencia [11] (componente eléctrico pasivo de dos terminales no linear faltante. Relaciona la vinculación de la carga eléctrica con un flujo magnético). Las células de estas memorias están formadas por dos capas de electrodos metálicos separados por un capa de un óxido de metal, como puede verse en la figura 4.1 [29], en la cual, la región con una vacante baja de oxígeno se comporta como un semiconductor, mientras que la otra región tiene una gran conductancia eléctrica (facilidad que presenta un conductor al paso de corriente eléctrica).

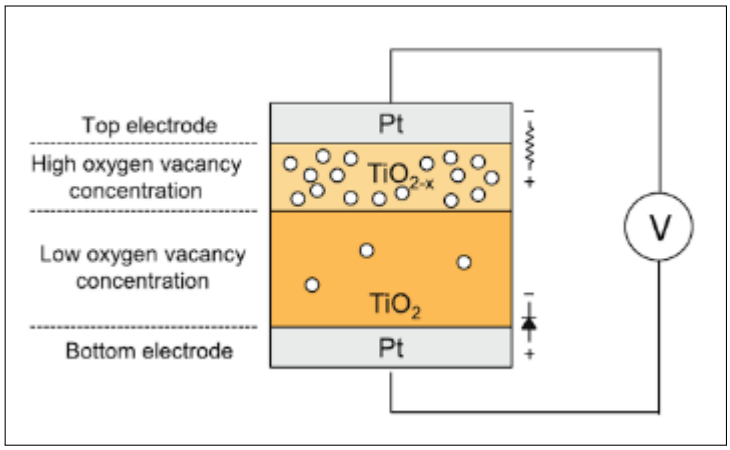

Figura 4.1: Célula de memoria RRAM

Su funcionamiento se basa en el cambio de la conductividad del dieléctrico, este se comporta en un principio como un aislante, pero tras aplicar un voltaje lo suficientemente alto permite conducir a través de un filamento que se forma a través de él entre los dos electrodos (este proceso se denomina forming o form como veremos en imágenes más adelante). Esta ruta entre los dieléctricos se puede formar mediante diferentes mecanismos, por vacantes o por migraciones

de defectos metálicos.

Una vez que se ha formado el filamento, se puede realizar un reset (ruptura de este filamento conductivo provocando una alta resistencia) o un set (reformando el filamento, baja resistencia) empleando para ello otro voltaje. Es posible la formación de múltiples rutas en lugar de un único filamento. Estas dos operaciones se emplean para representar un 1 (set) o un 0 (reset).

Este cambio entre reset y set de las memorias ReRAM puede ser:

- Unipolar: El cambio no depende de la polaridad del voltaje aplicado, pero generalmente este cambio ocurre con una amplitud del voltaje mayor que en el bipolar. Tienen una alta desuniformidad y pobre durabilidad (ciclos que soporta sin colapsar la ventana de memoria). Un ejemplo de este comportamiento puede observarse en la figura 4.2 [27].

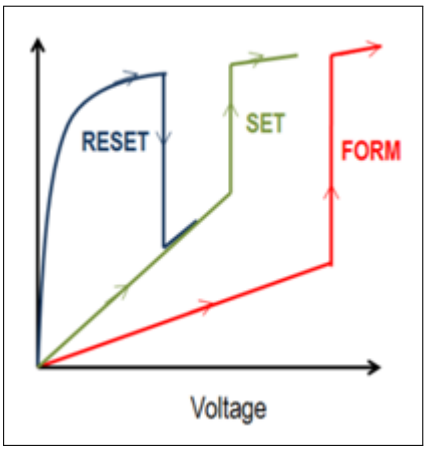

Figura 4.2: Comportamiento memoria RRAM unipolar

- Bipolar: El cambio entre el set y el reset depende de la polaridad. Un ejemplo de este comportamiento puede observarse en la figura 4.3 [27].

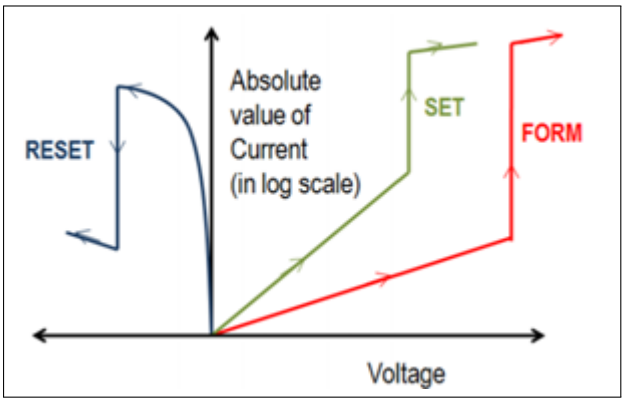

Figura 4.3: Comportamiento memoria RRAM bipolar

Como se puede observarse en las figuras 4.2 [27] y 4.3 [27], el proceso de forming requiere la aplicación de un mayor voltaje que el set.

Una estructura similar a la de la figura 4.1 [29] con un estado de resistencia inicialmente alto (IRS) puede cambiar a un estado de baja resistencia (LRS) mediante la aplicación de un alto voltaje de estrés. Este proceso se denomina *proceso de electroforming* o simplemente *proceso de forming* y altera la resistencia de la estructura. Tras el proceso de forming, el dispositivo RRAM puede ser devuelta a un estado de alta resistencia (HRS), el cual generalmente es menor que el IRS en el que se encontraba antes, mediante la aplicación de un voltaje concreto llamado voltaje de reset. Cuando se aplica el set, y se pasa de HRS a LRS es necesario limitar el voltaje aplicado para evitar dañar el dispositivo, como se vió en las figuras 4.2 [27] y 4.3 [27] el forming ocurre con un voltaje mayor que el set, y se produce un rápido crecimiento de la conductancia [4] (aparición de un número de filamentos o aumento de los existentes), pudiendo llegar al límite del dispositivo.

Mecanismos de cambio

En función de la ruta de conducción podemos diferenciar entre dos tipos de cambio:

1. Por filamento. El cambio se produce por la formación o ruptura del conducto de filamento en el material.

2. Por interfaz. El cambio se produce en la interfaz entre el metal y el aislante, encontramos varios métodos de implementación, como la migración de vacantes de oxígeno, la captura de portadores de carga (huecos o electrones) y una transición Mott [20] inducida por portadores dopados en la interfaz.

Mediante el comportamiento del dispositivo podemos saber ante el mecanismo de cambio que nos encontramos. Si la resistencia del LRS es independiente del área del dispositivo y el HRS varía inversamente, nos encontramos ante un mecanismo de cambio por filamento, en el caso en el cual ambos (LRS y HRS) aumentan con la reducción del área del dispositivo, se trata de un mecanismo de cambio por interfaz. Se pueden observar ambos tipos de cambio o conmutación en la figura 4.4 [21].

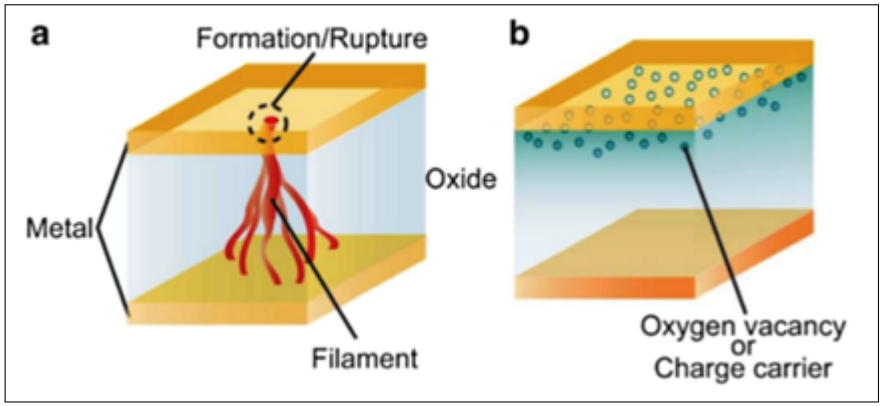

Figura 4.4: Conmutación por filamento (a) y conmutación por interfaz (b)

Además, teniendo en cuenta el material donde se produce el cambio y los electrodos, las memorias de cambio resistivo pueden ser divididas en dos tipos:

1. Conmutación basada en cationes o memoria de metalización electroquímica (ECM). Se emplea un electrolito sólido como material de conmutación, un material metálico electroquímicamente activo como electrodo superior y un metal inerte como electrodo inferior. Cuando se aplica un voltaje positivo a este electrodo superior se reduce electroquímicamente dando iones positivos, los cuales se difunden a través del sólido electrolito hasta alcanzar el electrodo inferior y electrocristalizar, generando un filamento conductor. Al aplicar un voltaje negativo al electrodo superior el filamento anteriormente formado se rompe.

2. Conmutación basada en aniones o memoria de cambio de valencia (VCM). Se emplea un material de conmutación subestequiométrico, un electrodo inerte y un electrodo reactivo. La conmutación ocurre debido a la reacción redox inducida por la migración de aniones para formar el filamento de conducción (figura 4.4 a [21]).

Requisitos

Las células RRAM deben cumplir una serie de requisitos [28]:

- Operación de escritura, el voltaje de escritura debe estar en el rango de unos pocos cientos de milivoltios a unos pocos voltios. Esto les da ventaja frente a las memorias Flash, las cuales sufren con altos voltajes de programación. El longitud del pulso de escritura suele ser de menos de 30 nanosegundos, lo cual permite que estas memorias compitan con las DRAM y tengan mejor rendimiento que las Flash.

- Operaciones de lectura, el voltaje para realizar las lecturas debe ser significativamente inferior que los voltajes de escritura, ya que si no es así se podría realizar una escritura sin ser esta la intención. Durante la toma de medidas en el laboratorio las lecturas se realizan a cero voltios.

- Ratio de resistencia, aunque tan solo un ratio de R_{OFF} / R_{ON} de aproximadamente 1.3 puede ser utilizado, ratios de 10 son necesarios para permitir el uso de Sense Amplifiers pequeños y de gran eficiencia, lo cual se traduce en chips RRAM con coste-efectividad razonables.

- Durabilidad, las memorias Flash actuales tienen un número máximo de ciclos de escritura de entre 10^3 y 10^7 (dependiendo del tipo de memoria flash), mientras que las memorias RRAM tienen el potencial de una durabilidad mucho mayor.

- Retención, para ser considerada una memorias no volátil, el periodo de retención de datos debe ser superior a 10 años. Durante este tiempo de retención de los datos debe poder soportar hasta temperaturas de 85ºC y pequeños pulsos eléctricos constantes.

Parte III

Simulador de memoria NVMain

Capítulo 5

Introducción a NVMain

Las memorias DRAM han proporcionado un baja latencia, un gran ancho de banda y un bajo consumo de energía, por ello han sido las memorias más empleadas y por lo tanto durante los últimos años se han desarrollado una gran cantidad de simuladores para llevar a cabo su estudio y mejora. El problema es que estos simuladores no son válidos para el estudio e investigación de otras tecnologías de memorias como las memorias no volátiles.

NVMain [25] [26] surge como alternativa a otros simuladores de memorias no volátiles los cuales habían sido realizados reutilizando simuladores de DRAM, por lo que no podían emular de manera adecuada el funcionamiento de estas memorias, ya que poseen características únicas para las que son necesarias, como un correcto modelado de la durabilidad, la recuperación de fallos y operaciones MLC (células multinivel) con alta exactitud en lo que se refiere a energía y latencia.

Actualmente nos encontramos en la versión 2.0 de este simulador, el cual incluye nuevas características y funcionalidades con respecto a su anterior versión para dar soporte a la simulación de memorias NVM y DRAM, esto se traduce en un nuevo sistema híbrido capaz de emular las memorias NVM y las DRAM, cada una con sus respectivos tiempos de respuesta, lo que requiere una mayor flexibilidad del simulador, así como un modelo del bank de memoria de grano fino, soporte MLC, mayor flexibilidad en la traducción de direcciones, y *hooks*, para que los usuarios puedan explorar nuevos diseños de sistemas de memoria.

5.1. Modelo energético

Los simuladores de DRAM emplean las medidas publicadas disponibles de los dispositivos actuales. Este enfoque es un problema cuando no se dispone de estos datos, algo que es común en el caso de las memorias no volátiles y en los diseños de prototipos de sistemas de memoria. Para evitar este problema NVMain proporciona dos dispositivos de modelos del nivel energético:

1. *Current-mode*: emplea los valores publicados disponibles de los dispositivos, se emplea para las simulaciones con DRAM estándar o para la parte de DRAM del sistema híbrido.

2. *Energy-mode*: emplea el uso de valores obtenidos mediante otras simulaciones como NV-Sim [14] o CACTI [3].

Además provee cálculos de energía independientes para los módulos restantes del subsistema de memoria. En las memorias no volátiles, el consumo energético durante las escrituras es mucho más elevado que para las lecturas

5.2. Arquitectura de memoria

En la figura 5.1 [1] se puede observar la arquitectura empleada por NVMain, en ella vemos como un rank posee varios device, cada device contiene varios banks con múltiples subarrays, los que están formados por diferentes MAT donde se encuentran las células, lugar en el cual se almacenan los bits. De esta manera se puede conseguir un paralelismo a nivel de subarray (SALP), los subarrays son seleccionados de la misma manera que otros objetos de memoria, empleando el traductor de direcciones, además contiene la escritura MLC, la durabilidad y el modelo de fallos.

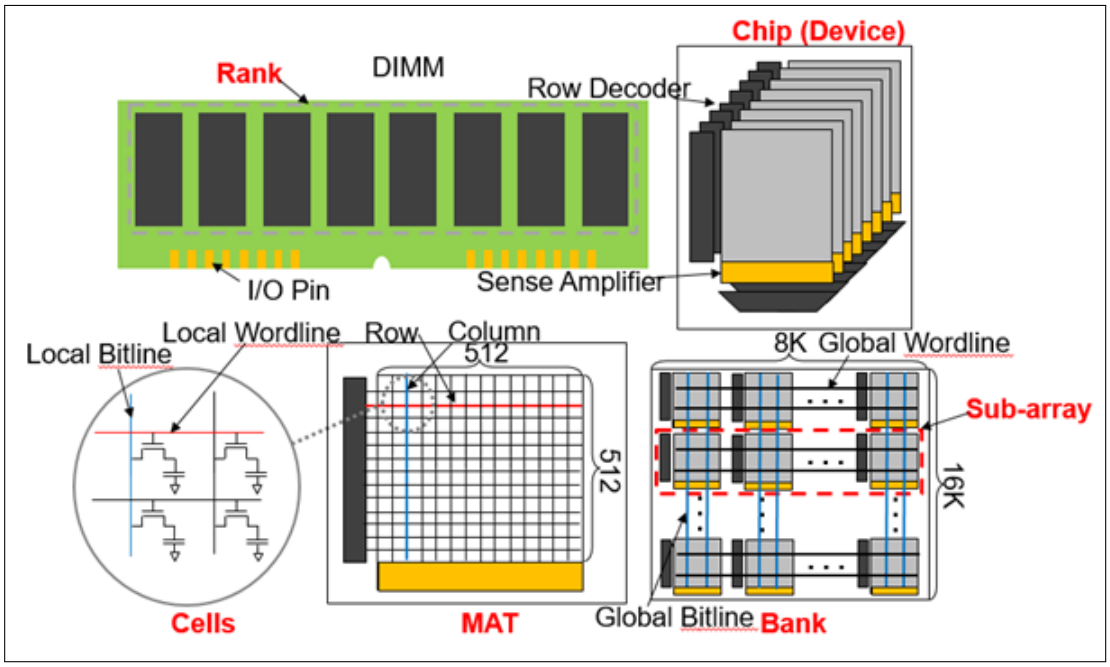

Figura 5.1: Arquitectura de NVMain

5.3. Restricciones temporales

A la hora de llevar a cabo las diferentes operaciones con NVMain nos encontramos con los retardos de tiempo asociados. Las operaciones que se verán afectadas por los retardos que van a describirse son:

- *PRE*: Precarga (precharging). En fig. 5.2 [1].
- *ACT*: Activación (activating). En fig. 5.2, 5.3 y 5.4 [1].
- *CAS*: Comando *CAS*, puede ser de lectura o de escritura. En fig. 5.2 y 5.5.

En función de los diferentes accesos necesarios encontraremos retardos [22] [23] [2] [10] a diferentes niveles, los cuales pueden ser:

- Intra-subarray, se corresponde con cualquier operación consecutiva que se lleve a cabo dentro del mismo subarray, como podemos ver en la figura 5.2 [1] los retardos asociados son:

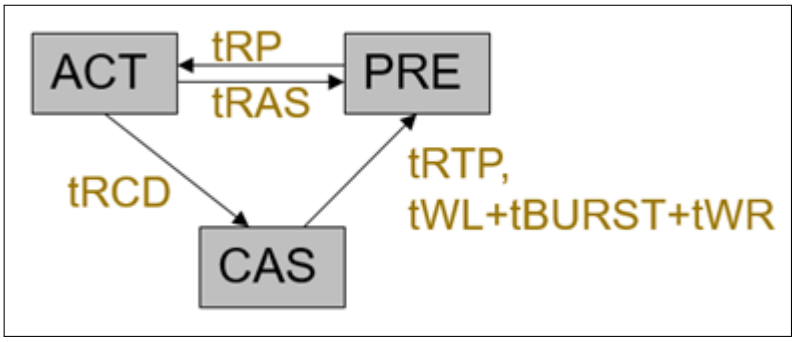

Figura 5.2: Retardos en intra-subarray

  - *tRAS*: tiempo de restauración de datos (tiempo transcurrido desde que se ejecuta un comando de activación hasta que se puede realizar una precarga).
  - *tRCD*: tiempo de activación (tiempo transcurrido entre un comando de activación y un comando de lectura o escritura).
  - *tRP*: tiempo de precarga (tiempo transcurrido entre un comando de precarga y un comando de activación).
  - *tRTP*: mínimo tiempo entre una lectura y una precarga.
  - *tWL*: latencia de escritura, tiempo después de una escritura hasta que el primer dato está disponible en el bus.
  - *tBURST*: tiempo de transmisión de los datos de manera continua (evitando la mayoría de los retardos por la sobrecarga del primer acceso a los demás).
  - *tWR*: tiempo de recuperación de escritura, mínimo tiempo después de una escritura hasta que una precarga se debe lanzar sobre el mismo bank.

- Intra-bank, se corresponde con cualquier operación consecutiva que se lleve a cabo dentro del mismo bank, cuyos retardos asociados se pueden observar en la figura 5.3 [1] y son:

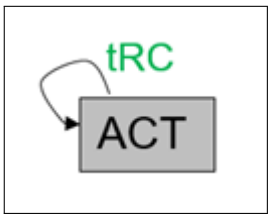

Figura 5.3: Retardos en intra-bank

- *tRC*: ciclo completo de lectura. Este ciclo es el menor tiempo posible para leer datos desde diferentes y aleatorias filas dentro de un subarray. Siempre y cuando se excluyan *tRRD* (cantidad de ciclos para activar el siguiente bank de memoria) y *tFAW* (mínimo intervalo de tiempo en el cual cuatro activaciones pueden ocurrir). Puede ser calculado mediante la fórmula:

```
tRC = tRCD + tRAS + tRP
```

- Intra-rank, se corresponde con cualquier operación consecutiva que se lleve a cabo dentro del mismo rank, el cual se puede ver en la figura 5.4 [1] junto con sus retardos asociados:

Figura 5.4: Retardos en intra-rank

  - *tRRD*: cantidad de ciclos para activar el siguiente bank de memoria.
  - *tFAW*: mínimo intervalo de tiempo en el cual cuatro activaciones pueden ocurrir.

- Intra-channel, se corresponde con cualquier operación consecutiva que se lleve a cabo dentro del mismo canal, el cual se encuentra en la figura 5.5 [1] junto con sus retardos:

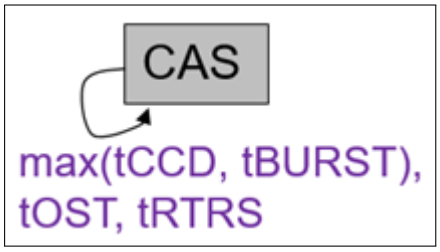

Figura 5.5: Retardos en intra-channel

  - *tCCD*: tiempo mínimo entre dos comandos de lectura o dos comandos de escritura.
  - *tBURST*: ya explicado anteriormente en intra-subarray.
  - *tOST*: tiempo para cambiar el control ODT entre ranks.
  - *tRTRS*: tiempo de cambio de un rank a otro.

Con respecto a estos retardos anteriores, es importante tener en cuenta que para memorias no volátiles, *tRAS* será igual a cero. Resulta relevante, que todos los parámetros que han sido descritos anteriormente son considerados antes de que cualquier comando de las memorias sea lanzado, y además, se toma siempre el valor del peor caso posible.

5.4. NMVain 2.0

Se introducen dos nuevos conceptos en NVMain 2.0, el traductor de direcciones avanzado y los objetos de memoria *hooks*. En primer lugar se cambia el flujo de las peticiones entre los objetos de memoria para invocar siempre al traductor de direcciones, después, se habilita el plugin de los objetos de memoria llamados *hooks*, los cuales pueden fisgonear peticiones al vuelo y cambiar el flujo de estas peticiones.

El traductor de direcciones recibe una dirección de memoria y extrae de esta unos valores concretos para con ellos determinar el bank destino de la petición. Los objetos de memoria *hooks*, son objetos de memoria externos, los cuales como se indicó anteriormente tienen la capacidad de fisgonear las peticiones de entrada o las de vuelta de un objeto de memoria concreto. Con respecto a otros objetos de memoria, debido a la naturaleza distribuida del diseño del simulador es posible la implementación de diferentes tipos de objetos de memoria que se pueden seleccionar en el tiempo de simulación, varios de estos objetos son proveídos por el propio NVMain. La razón de esto es la facilitación de la creación por parte de los usuarios de sus propios objetos de memoria.

NVMain implementa un sistema de memoria híbrido el cual emplea una migración de páginas hardware para poder demostrar la utilidad de un controlador de memoria avanzado y los *hooks* de memoria. Este sistema simulado emplea un controlador de memoria el cual usa una tabla de búsqueda para encontrar el destino de las páginas migradas e iniciar migraciones en consecuencia. Para facilitar la tarea, se utiliza una moneda parcial, para decidir si la página debe ser migrada o no. Básicamente, cada vez que se lanza una petición, hay una pequeña posibilidad por la cual se migre a otro canal de memoria, uno más rápido. Por ello, en teoría si una página recibe muchas peticiones, eventualmente será migrada a este canal de memoria rápida. En el caso del simulador NVMain, este canal de memoria rápida corresponde a DRAM y el resto de canales utilizan NVM como memoria lenta. Las migraciones se realizan utilizando el intercambio único de buffer.

5.5. Instalación de NVMain

Para compilar NVMain [13] en nuestra máquina se emplea *scons*. Por defecto *scons* compilará todos los ficheros objetivo del directorio actual donde se ejecuta o que se encuentren en subdirectorios. Además *scons* es capaz de detectar qué componentes han sido modificados y por lo tanto deben ser recompilados, ejecutando los comandos necesarios para ello. Es posible realizar una compilación nueva eliminando todos los archivos que se han generado anteriormente en otras compilaciones añadiendo el flag c".

Además NVMain permite trabajar de manera conjunta con *GEM5* [9], el cual es un simulador de plataforma modular para la investigación de la arquitectura de sistemas informáticos, *GEM5* engloba tanto la arquitectura de nivel de sistema, como la microarquitectura de procesadores. Por ello se empleará como un simulador de la CPU y de la caché. Desde NVMain se recomienda utilizar la versión más nueva de *GEM5* y no la versión estable, debido a que esta última es demasiado antigua. Para poder utilizar *GEM5*, es necesario *Mercurial* [12], un sistema gratuito

de control de versiones multiplataforma.

Es importante tener en cuenta que no es obligatorio utilizar NVMain con *GEM5*, en realidad puede emplearse cualquier otro simulador, solo que para poder conectar NVMain a cualquier simulador es necesaria una interfaz, la cual genera una petición *NMainRequest*. La interfaz necesita gestionar el envío de peticiones y manejar las respuestas. Normalmente, esta interfaz descrita anteriormente será el módulo *root* (raíz) de NVMain, lo que significa que se encargará de configurar el sistema entero de NVMain, además deberá registrar todos los *hooks* analizando el archivo de configuración. La interfaz debe heredar de la clase *NVMObject* para disponer del método *RequestComplete* con el que responderá a las peticiones.

Los pasos para instalar ambos paquetes de manera conjunta se describen a continuación (suponiendo que estamos como superusuario):

1. En primer lugar se instala *Mercurial*, *scons* y *g++*.

```
apt-get install mercurial scons g++
```

2. Se procede a descargar NVMain (es necesario solicitar acceso al repositorio).

```
hg clone https://bitbucket.org/mrp5060/nvmain/
```

3. Se compila NVMain. A la hora de realizar esta operación, puede ejecutarse el comando como se ve a continuación, o podemos incluir flags para seleccionar el tipo de compilación, si no incluimos ningún flag por defecto empleará *fast* la cual es una construcción del código completamente optimizada, las otras opciones son *debug* para depurar y *prof* que permite el soporte de perfiles. Los diferentes tipos de compilaciones se indican mediante *–build-type=*.

```
cd nvmain/
scons
```

4. Una vez se ha terminado la compilación se realiza una ejecución de prueba para comprobar que no ha habido ningún problema. NVMain se ejecuta con 3 parámetros de entrada, el archivo de configuración, el programa que ejecutará y el número de ciclos de CPU, como salida obtenemos los ciclos de la memoria que se han ejecutado junto con otros datos como puede verse en la figura 5.6.

```
./nvmain.fast Config/RRAM_ISSCC_2012_4GB.config Tests/Traces/
    hello_world.nvt 0
```

5. Se instalan los paquetes necesarios para poder instalar *GEM5*.

```
apt-get install swig python-dev libghc-zlib-dev m4
```

6. Se instalan los paquetes que permitirán obtener soporte de trazas.

```
i0.defaultMemory.totalReadRequests 49675
i0.defaultMemory.totalWriteRequests 48275
i0.defaultMemory.successfulPrefetches 0
i0.defaultMemory.unsuccessfulPrefetches 0
Exiting at cycle 2260015 because simCycles 0 reached.
```

Figura 5.6: Salida de la ejecución de NVMain

```
apt-get install protobuf-compiler libprotobuf-dev
```

7. Se instala un paquete para conseguir un mejor rendimiento con *GEM5*.

```
apt-get install libgoogle-perftools-dev
```

8. Se crea el archivo *.hgrc* en nuestro directorio *home* (principal) para que *Mercurial* lea la configuración y pueda gestionar parches como puede verse en la figura 5.7.

```
[extensions]
hgext.mq =
[ui]
username = Nombre Apellido <Correo@electronico>
```

Figura 5.7: Archivo .hgrc

9. Se inicializa *Mercurial* empleando:

```
hg init
```

10. Se configura una cola mediante:

```
hg qinit
```

11. Se aplica el parche a *GEM5* (partiendo del directorio padre de *GEM5*).

```
cd gem5
hg import -f ../nvmain/patches/gem5/nvmain2-gem5-11688+
hg qpush
```

12. Se comprueba si ha habido algún problema con el siguiente comando. Si todo ha sido correcto aparecerá el parche que se acaba de añadir.

```
hg qapplied
```

Capítulo 6

Características de NVMain

En este capítulo del documento se describirán los diferentes componentes del simulador, los parámetros de configuración que se emplean, su estructura y los objetos que los forman. En la figura 6.1 [1] puede verse representado el ecosistema completo de NVMain. En el capítulo siguiente se entrará mas en detalle en las partes que se van a estudiar en este documento, este capítulo consistirá en un pequeño resumen de la estructura del simulador NVMain para tener un mayor entendimiento del mismo necesario para los capítulos siguientes.

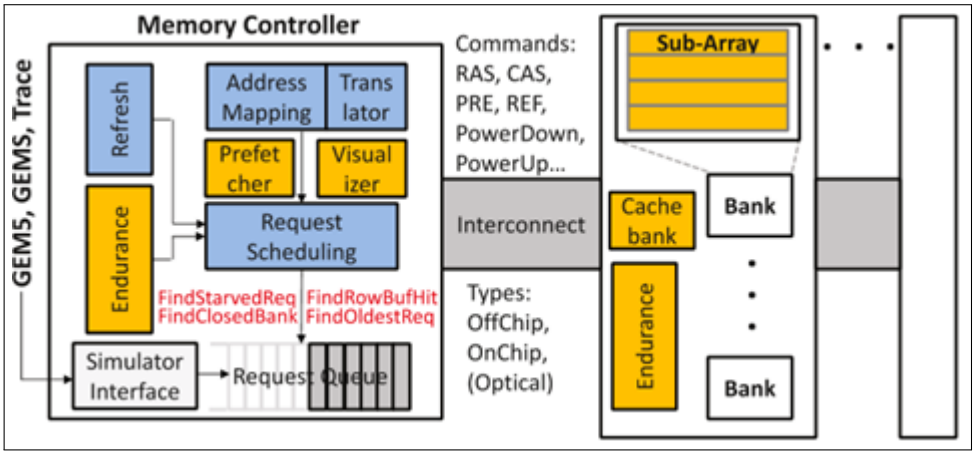

Figura 6.1: Visión global de NVMain

6.1. NVM Object

El sistema de memoria diseñado en NVMain es una abstracción en forma de árbol, con el objeto NVMain como raíz. Este objeto NVMain se conecta con los controladores de memoria, estos controladores están conectados al interconnect (el puente entre el controlador de memoria y la estructura de memoria). Este interconnect puede estar conectado a los rank, los cuales están conectados a los dispositivos, los cuales están conectados a los banks. Esta estructura de árbol que emplea NVMain puede verse representado en la figura 6.2.

Esta estructura de árbol tiene cada objeto siendo el padre de sus objetos hijo, solamente hay un

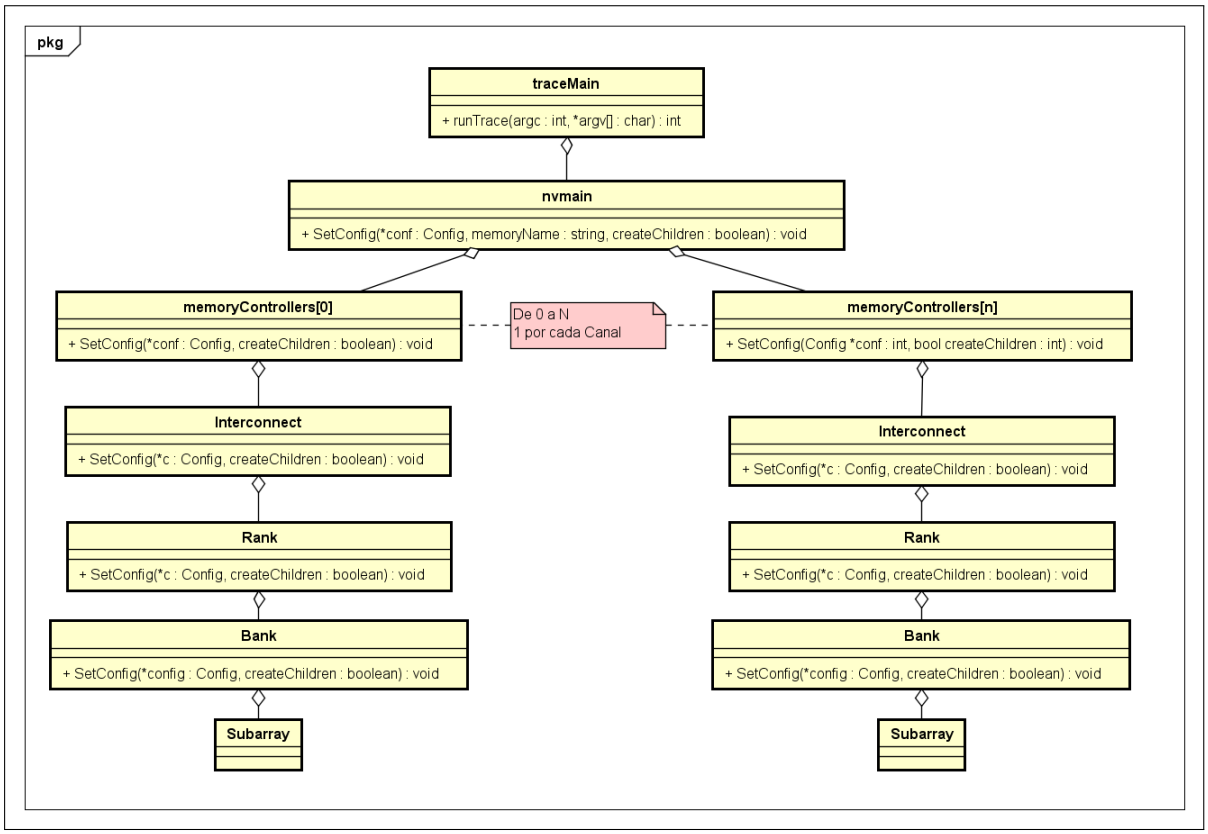

Figura 6.2: Estructura en árbol de NVMain

padre y múltiples hijo. Cuando las peticiones son propagadas a través del simulador, las peticiones entrantes utilizan un decoder para seleccionar el objeto hijo. Por lo general este decoder es un traductor de direcciones. Una vez que las peticiones han sido servidas, generalmente por banks en el punto final, la respuesta se propaga a través de los padres hasta que se alcanza el propietario de la petición. Al construir un sistema de memoria con un objeto genérico *NVMObject*, el padre y los hijos deben ser establecidos. *NVMObject* también tiene una interfaz genérica, para obtener estadísticas de cada módulo, estas estadísticas incluyen área, energía, retardos, etc.

En la estructura en árbol de la figura 6.2 se han incluido en las operaciones los métodos en los cuales se crea el objeto hijo y se añade a la estructura mediante *AddChild()*. Con respecto al objeto subarray, solamente se crea si se utiliza como clase bank *DDR3Bank* (por defecto si no se indica lo contrario), si se emplea *CacheDDR3Bank* este es el último elemento hijo.

6.2. Decoders

El trabajo del decoder es tomar una dirección de memoria y enrutar la petición a la localización de memoria correcta. El decoder más simple solamente miraría los bits en varias posiciones y enrutaría la petición en función de eso. El orden de desciframiento es “C:H:K:B:R” siendo estos valores:

1. C, columna.

2. H, canal.
3. K, rank.
4. B, bank.
5. R, fila.

Esto significa que la columna es escogida empleando los bits menos significativos, seguida por el canal, el rank, el bank y la fila en los bits más significativos. Otros diseños de memoria pueden requerir decoders más complejos. Por ejemplo, en un sistema de memoria de caché de una escala de gigas se puede renunciar al campo de columna, en lugar de ello optando por proveer el conjunto donde reside la dirección de memoria caché, y una etiqueta para verificar el acierto.

Cualquier esquema de decoder puede ser especificado mediante los parámetros de configuración de *AddressMappingScheme*. Para ello se emplea C para columna, R para fila, SA para subarray, BK para bank, RK para rank y CH para canal. A continuación se muestran unos ejemplos:

- SA:R:RK:BK:CH:C, prioriza los accesos a la misma fila del buffer (opción por defecto), peticiones a una misma fila del buffer.
- SA:R:RK:BK:C:CH, prioriza el intercalado de canales, peticiones a diferentes canales.
- SA:R:RK:CH:C:BK, prioriza el intercalado de banks, peticiones a diferentes banks.
- SA:R:BK:CH:C:RK, prioriza el intercalado de ranks, peticiones a diferentes ranks.

6.3. Controladores de memoria

Para crear un nuevo controlador de memoria para NVMain no es necesario tener conocimiento sobre los retardos de los banks de memoria, además se proveen ciertas funciones de programación para que sea más sencillo.

El controlador de memoria provee algunas funciones comunes que se pueden utilizar cuando se programan peticiones. Estas funciones hacen uso de las funciones *IsIssuable* e *IssueCommand*, las cuales son requeridas por todos los NVMObjects, por lo que a la hora de crear nuestros controladores de memoria no tenemos que usar obligatoriamente las siguientes funciones porque se puede esperar a que una petición sea expedible.

Tanto los controladores de memoria expuestos en esta parte como las funciones comunes serán explicadas en más detalle más adelante en el documento.

6.3.1. Funciones comunes

A continuación se listan y describen estas funciones de programación comunes de las que se habló anteriormente:

- *FindStarvedRequest*, examina la cola de transacciones para detectar si se ha dado el caso que ha habido muchas peticiones exitosas a una misma fila del buffer, las cuales han causado que una petición se retrase y devuelve esa petición para evitar la inanición.

- *FindRowBufferHit*, examina la cola para encontrar una petición que pueda ser servida sin tener que ejecutar un comando de activación o precarga a la memoria (petición con objetivo a la misma fila del buffer disponible en los sense amplifiers).
- *FindCacheAddress*, busca una dirección que pueda ser leída o escrita en algún lugar de la memoria sin enviar un comando de precarga o de activación.
- *FindOldestReadyRequest*, encuentra la petición más antigua en la cola de transacciones que está lista para ser servida. Como se asume que la cola de transacciones está ordenada, simplemente empieza buscando desde el principio de la cola hasta el final.
- *FindCloseBankRequest*, busca una petición a un bank que no está activo, la primera petición emitida y las peticiones que lleguen tras una apagado serán servidas por esta función.
- *IssueMemoryCommands*, toma una petición y emite el comando correspondiente a la memoria principal.

6.3.2. Controladores iniciales

Desde NVMain se proveen además varios controladores de memoria como ejemplo para facilitar la modificación/creación de controladores nuevos, los cuales se describirán a continuación.

FCFS

Primero en llegar, primero en ser servido, este es un ejemplo muy sencillo de un controlador de memoria, las peticiones son servidas en el orden en el que llegan al controlador, se ignora el rank o el bank al que se deben dirigir. La petición más antigua se puede encontrar con la función *FindOldestReadyRequest* y se emite. Si se necesitan activaciones, se pueden crear de manera automática mediante la función *IssueMemoryCommands*. Con respecto a este controlador, si queremos modificar su comportamiento, debemos modificar los siguientes elementos en el código que son los que establecen su configuración:

1. *QueueSize*, establece el número máximo de peticiones de lectura o escritura en la cola de transacciones.
2. *MEM_ CTL*, establece el controlador de memoria a utilizar, este se corresponde con FCFS.

Por lo tanto, en este controlador de memoria, desde el bucle del programa principal se pregunta si es utilizable (*IsIssuable*), en caso de ser utilizable significa que la cola de transacciones no está llena y se puede añadir la petición, por lo que se lanza este proceso (*IssueCommand*), en este método se vuelve a comprobar que la cola de transacciones este disponible antes de proceder y la incluye en la cola de transacciones (método *Enqueue* en *src/MemoryController.cpp*), tras esto comprueba si la cola de comandos está vacía, en cuyo caso añade una petición de *wake up* en el ciclo actual a la lista de eventos.

Para ejecutar estas peticiones, el proceso se realiza desde el método *Cycle*, como ya se indicó antes, se busca la petición (en la cola de transacciones) más antigua, si no hay ninguna busca una

petición a un bank cerrado. Cuando tiene una petición, obtiene el comando a emitir a la memoria principal empleando el método *IssueMemoryCommands*, en este método las peticiones pasan de la cola de transacciones a la cola de comandos, tras esto con el método *CycleCommandQueues* elimina la petición de la cola de comandos y la añade a la lista de eventos.

La resolución final de las peticiones se realiza cuando, desde el bucle principal en el cual se van avanzando ciclos se termina por llegar al método *Process* (objeto *EventQueue.cpp*) ejecutando el método *Cycle* del objeto *globalEventQueue* (En este método se ejecutan otros métodos relevantes como *GetNextEvent*, *Loop* y *Process*).

FR-FCFS

Primera lista, primera en llegar, primera en ser servida, se trata de una versión más complicada del ejemplo anterior, en este caso busca las peticiones que se realizarán a una misma fila del buffer para reducir latencias. Si no encuentra peticiones a una misma fila del buffer, busca la petición lista más antigua o la petición a un bank cerrado. Finalmente, comprueba si se ha llegado al estado de inanición antes de buscar otras peticiones. Una vez que alguno de los comandos encuentra una petición, las funciones restantes son saltadas y la primera petición encontrada es servida. Este controlador es el más usado en investigación, sin embargo puede haber diversas implementaciones de este controlador de memoria. La implementación de ejemplo funciona de la siguiente manera:

1. Comprueba si hay peticiones esperando por el recurso, si las encuentra las sirve inmediatamente.

2. Comprueba si hay peticiones a la misma fila del buffer, si las encuentra las prioriza sobre las otras peticiones.

3. Busca la petición lista más antigua y la emite.

4. Si no hay peticiones listas no hace nada.

Los parámetros de configuración relaciones con este controlador son:

1. *QueuseSize*, establece el número máximo de peticiones de lectura o escritura en la cola de transacciones.

2. *StarvationThreshold*, establece el número máximo de peticiones a la misma fila de buffer que pueden ser servidas antes de que una petición más antigua del mismo bank tenga prioridad.

Por lo tanto, en este controlador de memoria, desde el bucle del programa principal se pregunta si es utilizable (*IsIssuable*), en caso de ser utilizable significa que la cola de transacciones no está llena y se puede añadir la petición, por lo que se lanza este proceso (*IssueCommand*), en este método se vuelve a comprobar que la cola de transacciones este disponible antes de proceder y la incluye en la cola de transacciones (método *Enqueue* en *src/MemoryController.cpp*), tras esto

comprueba si la cola de comandos está vacía, en cuyo caso añade una petición de *wake up* en el ciclo actual a la lista de eventos.

Para ejecutar estas peticiones, el proceso se realiza desde el método *Cycle*, como ya se indicó antes, se busca en primer lugar peticiones que estén en situación de inanición, tras esto peticiones a una misma fila del buffer, peticiones disponibles en caché, peticiones de escrituras pausadas para lanzar peticiones de lectura, la petición más antigua o peticiones a un bank cerrado. Cuando tiene una petición, obtiene el comando a emitir a la memoria principal empleando el método *IssueMemoryCommands*, en este método las peticiones pasan de la cola de transacciones a la cola de comandos, tras esto con el método *CycleCommandQueues* elimina la petición de la cola de comandos y la añade a la lista de eventos.

La resolución final de las peticiones se realiza cuando, desde el bucle principal en el cual se van avanzando ciclos se termina por llegar al método *Process* (objeto *EventQueue.cpp*) ejecutando el método *Cycle* del objeto *globalEventQueue* (En este método se ejecutan otros métodos relevantes como *GetNextEvent*, *Loop* y *Process*).

FR-FCFS with Write-Buffering

Con este controlador conseguimos priorizar las lecturas sobre las escrituras, esto se consigue empleando un buffer de escritura. Las peticiones de escritura se insertan en una cola de transacciones de escritura, y las lecturas se emiten a una cola de transacciones de lectura. Todas las funciones comunes permiten que se les pase un predicado, este predicado devolverá true cuando el buffer de escritura esté lleno, por lo tanto, el controlador podrá usar las mismas funciones que emplea el controlador de memoria FR-FCFS, solo que añadiendo el predicado para realizar estas peticiones contra la cola de escritura. Con respecto a la cola de lectura, el controlador emplea las funciones comunes con FR-FCFS sin emplear el predicado. Cuando el predicado devuelve True, lo que significa que el buffer de escritura está lleno como se indicó anteriormente, se llevará a cabo un drenado del mismo para servir estas peticiones.

Por ello, el controlador FR-FCFS with Write-Buffering es muy similar al controlador FR-FCFS pero empleando dos colas (lectura y escritura). Como se ha dicho anteriormente, se priorizan las lecturas sobre las escrituras, hasta que se alcanza un límite superior establecido (llamado *high-water mark*), momento en el cual el controlador de memoria drena la cola de escritura hasta que se alcanza el límite inferior establecido (*low-water mark*). Entonces mientras se está ejecutando la simulación, estas dos marcas servirán para alternar entre las dos colas de peticiones. La interacción mientras se está drenando el buffer de escritura es:

1. Comprobar si hay peticiones listas que esperan por recursos, en ese caso se sirven inmediatamente.

2. Comprobar si hay una petición de escritura a la misma fila del buffer, en cuyo caso se prioriza sobre las otras peticiones.

3. Encontrar la petición más antigua en la cola y servirla.

4. Si no hay peticiones listas no se hace nada.

La interacción mientras se están sirviendo las peticiones de la cola de lectura es:

1. Comprobar si hay peticiones listas que esperan por recursos, en ese caso se sirven inmediatamente.
2. Comprobar si hay una petición de lectura a la misma fila del buffer, en cuyo caso se prioriza sobre las otras peticiones.
3. Encontrar la petición más antigua en la cola y servirla.
4. Si no hay peticiones listas no se hace nada.

Los parámetros de configuración relacionados con este controlador son:

1. *ReadQueueSize*, número máximo de peticiones en la cola de transacciones de lectura.
2. *WriteQueueSize*, número máximo de peticiones en la cola de transacciones de escritura.
3. *StarvationThreshold*, número máximo de peticiones a una misma fila del buffer que se pueden priorizar antes de que una petición más antigua al mismo bank tenga prioridad.
4. *HighWaterMark*, número de peticiones en la cola de escritura a partir del cual se iniciará el drenaje.
5. *LowWaterMark*, número de peticiones en la cola de escritura a partir del cual se detendrá el drenaje.

Por lo tanto, en este controlador de memoria, desde el bucle del programa principal se pregunta si es utilizable (*IsIssuable*), en este caso lleva a cabo una comprobación en función del tipo de petición entrante con respecto a su cola correspondiente indicando que no es utilizable si la cola no tiene espacio libre, en caso de tratarse de una petición de escritura esta no va a poder lanzarse si se está en proceso de drenaje, hasta que el drenaje no finalice no será incluida en la cola. En caso de ser utilizable significa que la cola de transacciones no está llena y se puede añadir la petición, por lo que se lanza este proceso (*IssueCommand*), en este método se vuelve a comprobar que la cola de transacciones este disponible antes de proceder y la incluye en la cola de transacciones (método *Enqueue* en *src/MemoryController.cpp*), tras esto comprueba si la cola de comandos está vacía, en cuyo caso añade una petición de *wake up* en el ciclo actual a la lista de eventos.

Para ejecutar estas peticiones, el proceso se realiza desde el método *Cycle*, como ya se indicó antes, primero lleva a cabo las comprobaciones sobre si se debe iniciar o acabar el drenaje y tras esto, ya sea para lectura o para escritura, se busca en primer lugar peticiones que estén en situación de inanición, tras esto peticiones a una misma fila del buffer, peticiones disponibles en caché, peticiones de escrituras pausadas para lanzar peticiones de lectura, la petición más antigua o peticiones a un bank cerrado. Cuando tiene una petición, obtiene el comando a emitir a la memoria principal empleando el método *IssueMemoryCommands*, en este método las peticiones pasan de la cola de transacciones a la cola de comandos, tras esto con el método *CycleCommandQueues* elimina la petición de la cola de comandos y la añade a la lista de eventos.

La resolución final de las peticiones se realiza cuando, desde el bucle principal en el cual se van avanzando ciclos se termina por llegar al método *Process* (objeto *EventQueue.cpp*) ejecutando el método *Cycle* del objeto *globalEventQueue* (En este método se ejecutan otros métodos relevantes como *GetNextEvent*, *Loop* y *Process*).

PerfectMemory

Representa a un controlador de memoria perfecto, el cual devuelve al simulador de manera inmediata los resultados de las peticiones y actúa como si la latencia para todas estas peticiones fuera cero.

No dispone de parámetros de configuración, ya que no es necesario establecer ningún tipo de prioridad entre colas, indicar el tamaño de la cola de transacciones o priorizar un tipo de peticiones sobre otras, pues se sirven de manera inmediata.

6.4. Durabilidad

Los modelos de durabilidad se utilizan para proveer tolerancia a fallos en las memorias no volátiles en el caso de atascamiento en fallos. La interfaz del modelo de durabilidad intercepta las peticiones de lectura a los banks y crea un contador con el número de escrituras a una región concreta de la memoria, esta región puede variar desde una página completa hasta un único bit. Los contadores de cada región son creados utilizando una distribución de probabilidad, cuando uno de estos contadores alcanza 0, la región se considera como atascada en algún valor y entra en juego el modelo de fallos.

NVMain provee varios ejemplos de modelos de durabilidad y dos distribuciones, estos diferentes modelos de durabilidad llevan a cabo el seguimiento descrito anteriormente para diferentes tamaños de regiones. Estos modelos de durabilidad no deben ser usados en las simulaciones a no ser que sea lo que se desea estudiar, ya que consumen una gran cantidad de recursos y son muy lentos, por lo tanto no es aconsejable su uso si no es el objetivo de los experimentos y se debe especificar *NullModel* en la configuración.

Cuando se modela la durabilidad, la media y la varianza de la distribución debe ser relativamente pequeña para asegurarse que los errores ocurren en un tiempo razonable de simulación, además de esto el sistema de memoria debe ser pequeño, ya que la cantidad de memoria que se utiliza para este rastreo puede llegar a ser muy grande, por ejemplo, en un sistema de rastreo de regiones del tamaño de un bit en un sistema de 128 MB requiere 8 GB.Los diferentes modelos de durabilidad disponibles son:

- *NullModel*, se asume durabilidad infinita, no se lleva a cabo ningún rastreo y no se utiliza almacenamiento para ello.

- *FlipWrite*, se trata del ejemplo de una propuesta real de un esquema de protección de la durabilidad, el cual almacena el valor de la palabra o el valor de la palabra invertido, en función de cual tiene la menor distancia hamming [6]. Adicionalmente, el bit inverti-

do es almacenado. El rastreo se realiza a nivel de bit, por lo tanto esto requiere mucho almacenamiento.

- *RowModel*, es un modelo de durabilidad de una fila completa, una vez que un número concreto de escrituras se realiza sobre una fila, esta fila completa se considera como atascada en un valor.
- *WordModel*, es un modelo de durabilidad de una palabra completa (el tamaño de un burst de columna, generalmente una línea de caché), cuando un número concreto de escrituras se realiza en esta palabra, la palabra completa se considera atascada en un valor.
- *ByteModel*, es un modelo de durabilidad a nivel de byte, cuando se produce un número concreto de escrituras en el byte, este byte completo se considera como atascado en un valor.
- *BitModel*, es un modelo de durabilidad a nivel de bit, cuando se realiza un número concreto de escrituras sobre un bit, este se considera como atascado en un valor.

Los parámetros de configuración referidos al modelo de durabilidad son:

1. *EnduranceModel*, indica el modelo de durabilidad que se utilizará.
2. *FlipWriteGranularity*, especifica el tamaño de una palabra en el modelo *FlipNWrite*.

Anteriormente se habló sobre las distribuciones de durabilidad, contamos con 2 tipos:

1. Normal, se trata de una distribución gausiana [7] que da el valor a estos contadores de durabilidad para cada región.
2. Uniforme, se emplea el mismo valor para todos los contadores de durabilidad en cada región.

Contamos además con parámetros de configuración referidos a las distribuciones de durabilidad:

1. *EnduranceDist*, indica la distribución de durabilidad que se utilizará.
2. *EnduranceDistMean*, indica la media tanto para las distribuciones normales como para las uniformes.
3. *EnduranceDistVariance*, indica la varianza para los valores de los contadores en una distribución normal.

6.5. Interconnect

El trabajo del interconnect es conectar el controlador de memoria al subsistema de memoria. Se asume que el controlador de memoria está en la raíz, para una simulación de CPU el controlador de memoria estaría dentro de la CPU. Si este no es el caso, un interconnect se debe situar entre el controlador de memoria principal y el controlador de memoria real reenviando las peticiones.

Crear un interconnect es relativamente sencillo, hay que hacer uso al menos de los métodos *IssueCommmand* y *RequestComplete*, para un interconnect más avanzado, *IsIssuable* puede ser también necesario para evitar enviar peticiones a través del interconnect.

Para un simple cable de bus *IssueCommand* simplemente enviará la petición al hijo o a los hijos que estén conectados al cable. Cuando se completa la petición, *RequestComplete* puede ser utilizado para enviar la respuesta desde el hijo hasta el padre. En ambos casos, la energía utilizada puede ser calculada basándose en los datos que hay en la petición.

En interconnects más complejos, puede ser necesario el uso de *IsIssuable*, si no hay capacidad para enviar la petición, *IsIssuable* devolverá falso, con ello se evita enviar objetos al módulo aunque el subsistema de memoria hijo no esté ocupado.

6.6. Prefetchers

El prefetching en memoria se provee en el módulo raíz de NVMain. Este prefetching se realiza ahí para asegurar que las peticiones pueden ser encoladas en los controladores de memoria y son direccionadas a diferentes canales de manera adecuada. Aunque es posible realizar este prefetching en otras zonas dentro de la jerarquía de memoria, también es mucho más complicado, el prefetching descrito anteriormente consiste en dos métodos primarios *NotifyAccess* y *DoPrefetch*.

Ante las peticiones de llegada, la clase NVMain llama a un método del prefetcher, *NotifyAccess*, este método se debe utilizar para predecir patrones y evaluar la eficiencia del prefetcher. Si se notifica un acceso y anteriormente fue prefetched, de esta manera el prefetcher sabe que el prefetch fue efectivo.

El método *DoPrefetch* es llamado en cada acceso. Si se debe realizar un prefetch, este método devolverá true y una lista de direcciones que deben ser prefetched. La clase NVMain automáticamente generará peticiones y tratará de encolarlas. Cualquier petición que no pueda ser encolada (porque el controlador de memoria no esté disponible por ejemplo) será descartada.

Las peticiones que se han completado por el prefetcher son colocadas en un buffer de prefetch, cualquier petición que llega y es encontrada en el buffer de prefetch es servida inmediatamente y se desaloja del buffer. Si el buffer está lleno la petición más antigua es eliminada. El tamaño del buffer de prefetch puede ser controlado utilizando el parámetro *PrefetchBufferSize*. Las peticiones que son desalojadas del buffer de prefetch sin haber sido servidas son marcadas como prefetches fracasadas por el módulo de estadísticas *succesfulPrefetches*. Las peticiones que han sido servidas por el buffer de prefetch son consideradas exitosas por el módulo de estadísticas *succesfulPrefetches*.

6.7. Archivo de configuración

Se tomará como base el archivo de ejemplo de configuración de NVMain disponible en *nvmain/Config* sobre RRAM, para entender los parámetros que emplea y poder modificarlo para que se adapte a nuestras simulaciones. Pero primero es importante entender los parámetros que utiliza y sobre los cuales ya se ha hablado brevemente realizando menciones. Podemos encontrar

los siguientes parámetros en el archivo de configuración referidos a la arquitectura de la memoria, especificaciones de la interfaz y la configuración general del sistema de la memoria:

- *CLK*: la frecuencia de reloj de la memoria, en megahercios.
- *RATE*: tipo de transferencia de datos, *SDR* (Single Data Rate) y *DDR* (Double Data Rate).
- *BusWidth*: ancho del bus, en bits.
- *DeviceWidth*: número de bits proporcionados por cada dispositivo en un rank. El número total de dispositivos se calcula como *BusWidth* / *DeviceWidth*.
- *CPUFreq*: la frecuencia de la CPU.
- *BANKS*: número de banks por rank.
- *RANKS*: número de ranks por canal.
- *CHANNELS*: número de canales en el sistema.
- *ROWS*: número de filas en un bank.
- *COLS*: número de columnas visibles en un bank. (MATs en cada fila del BANK)
- *MATHeight*: número de filas de cada MAT. Aquí podemos establecer el nivel de paralelismo a nivel de subarray, por ello:
  - *MATHeight* $<$*ROWS*, hay paralelismo a nivel de subarray, multiples subarrays. Ya que el número de subarrays es igual a *ROWS* / *MATHeight*
  - *MATHeight* = *ROWS*, no hay paralelismo a nivel de subarray, solo un subarray.
- *RBSize*: Referencia al multiplexor que se empleará, mediante el cual se podrán calcular los elementos del subconjunto disponibles que se cargan en el sense amplifiers de la fila de memoria objetivo.
- *UseRefresh*: indica si es necesario o no realizar un refresco de la información de la memoria (true o false). En este caso hay tres valores referidos al refresco, no se utilizan, pero igualmente indica que se les asignarán valores válidos, como no son relevantes para la simulación no se explicarán *BanksPerRefresh*, *RefreshRows* y *DelayedRefreshThreshold*.

Además el archivo de configuración contiene parámetros de retardo del dispositivo de memoria, los cuales son:

- *tBURST*: tiempo de transmisión de los datos de manera continua (evitando la mayoría de los retardos por la sobrecarga del primer acceso a los demás).
- *tCMD*: duración de transporte de comando, es el periodo de tiempo en el cual el comando ocupa el bus de comandos mientras es transferido desde el controlador hasta el dispositivo.

- *tRAS*: tiempo de restauración de datos (tiempo transcurrido desde que se ejecuta un comando de activación hasta que se puede realizar una precarga).
- *tRCD*: tiempo de activación (tiempo transcurrido entre un comando de activación y un comando de lectura o escritura).
- *tWP*: tiempo de escritura de un pulso.
- *tRP*: tiempo de precarga (tiempo transcurrido entre un comando de precarga y un comando de activación).
- *tCAS*: latencia estroboscópica de acceso a columna.
- *tAL*: latencia añadida al acceso de columna.
- *tCCD*: tiempo mínimo entre dos comandos de lectura o dos comandos de escritura.

Los siguientes parámetros se corresponden con retardos debidos a los circuitos de control:

- *tCWD*: retardo de escritura de columna, es el intervalo de tiempo desde que se lanza un comando de escritura en una columna y se colocan los datos en el bus de datos por el controlador.
- *tWTR*: retardo del comando de escritura interna para leer.
- *tWR*: tiempo de recuperación de escritura, tiempo después de una escritura hasta que se puede realizar una precarga al mismo bank.
- *tRTRS*: tiempo de cambio de un rank a otro.
- *tRTP*: mínimo tiempo entre una lectura y una precarga.
- *tOST*: tiempo para cambiar el control ODT entre ranks.

Los parámetros *tRRDR*, *tRRDW*, *RAW* y *tRAW* sirven para asegurar la integridad de los datos. Para tener en cuenta los retardos causados por el inicio y el fin del apagado, se emplean los siguientes parámetros.

- *tRDPDEN* [5]: retardo tras un comando de lectura antes de que se pueda lanzar un comando de apagado sobre el mismo rank. ($tCAS + tBURST$).
- *tWRPDEN*: retardo tras un comando de escritura antes de que se pueda lanzar un comando de apagado sobre el mismo rank ($tAL + tCWD + tBURST + tWP$).
- *tPD*: tiempo desde que se lanza el apagado hasta que se completa.
- *tXP*: tiempo de encendido partiendo del apagado.
- *tXPDLL*: tiempo para que se produzca un apagado profundo de la memoria.

Los siguientes parámetros corresponden a retardos de tiempo debidos al refresco de las memorias, aunque no afectan a las ReRam, en el fichero de ejemplo les asigna valores válidos:

- *tRFC*: tiempo entre un comando de refresco y un sucesivo comando de refresco o de activación.
- *tREFW*: tiempo de refresco de ventana.

NVMain también tiene en cuenta la energía consumida y utilizada por los dispositivos que simula, entre ellos encontramos *Eopenrd*, *Epdpf*, *Epdps*, *Epda* y *Eref*, estos parámetros se obtienen de *NVSIM* y *CACTI*, junto con ellos emplea los siguientes parámetros que se describen más en detalle:

- *Erd*: energía de lectura a una única matriz.
- *Ewd*: es la energía empleada para la escritura, da igual set que reset.
- *Ewrpb*: energía de escritura de un subarray por bit.
- *Eleak*: energía que se filtra en 1 segundo.
- *Voltage*: El voltaje que se aplica.

Además encontramos los parámetros de configuración del controlador de memoria:

- *MEM_ CTL*: selecciona el controlador de memoria que se va a utilizar.
- *ClosePage*: política de gestión del cierre de páginas del buffer de filas, podemos seleccionar entre tres diferentes:
  - 0, se cerrará la fila hasta que ocurra un fallo de buffer de fila.
  - 1, la fila se cerrará si no hay otra petición al buffer de la fila.
  - 2, la fila se cierra de forma inmediata, no se puede utilizar para otra petición al buffer de la fila (se deberá volver a abrir).
- *ScheduleScheme*: Esquema de la programación de los comandos, en este caso también contamos con tres posibilidades:
  - 0, prioridadfija.
  - 1, primero los ranks empleando round-robin.
  - 2, primero los banks empleando round-robin.
- *AddressMappingScheme*: esquema del mapeo de direcciones.
- *INTERCONNECT*: interconexiones entre los chips de memoria y los controladores.

El siguiente parámetro es un parámetro propio del controlador de memoria, en este caso referencia a FCFS:

- *QueueSize*: tamaño de la cola.

Para el control de la simulación se emplean los siguientes parámetros:

- *PrintGraphs*: Tan solo toma su valor sin emplearlo, posible futuro módulo de NVMain.
- *PrintPreTrace*: Relacionado con los parámetros de este subconjunto, si se define como true trata de leer el archivo de trazas de entrada (*PreTraceFile*) con el lector de trazas indicado (*TraceReader*).
- *PreTraceFile*: Sirve para indicar el archivo de trazas desde el archivo de configuración.
- *EchoPreTrace*: Relacionado con los parámetros de este subconjunto, si se define como true trata de leer el archivo de trazas de entrada (*PreTraceFile*) con el lector de trazas indicado (*TraceReader*).
- *PeriodicStatsInterval*: Tan solo toma su valor sin emplearlo, posible futuro módulo de NVMain.
- *TraceReader*: el objeto que se empleará para leer el archivo de trazas.

Los parámetros del modelo de durabilidad son:

- *EnduranceModel*: el modelo de durabilidad que se empleará.
- *EnduranceDist*: indica la distribución de durabilidad que se utilizará.
- *EnduranceDistMean*: indica la media tanto para las distribuciones normales como para las uniformes.
- *EnduranceDistVariance*: indica la varianza para los valores de los contadores en una distribución normal.
- *InitPD*: establece la DRAM en modo apagado al empezar (true), o no (false).

6.8. Archivo de trazas

NVMain emplea como entrada para el simulador los archivos de traza con extensión *.nvt*, los cuales cuentan con cinco campos, los cuales están especificados por la siguiente estructura:

```
1 ciclo   tipoDeOperación   direcciónDeMemoria   datos(64 bytes)   idHilo
```

Esta estructura descrita puede observarse en la figura 6.3, se describen los diferentes campos de la siguiente manera como puede verse:

- *ciclo*, ciclo de llegada de la petición, en formato decimal. Corresponde con los ciclos de procesador ejecutados.
- *tipoDeOperación*, se trata de un carácter el cual puede ser:

1. R, para indicar una petición de lectura.
2. W, para indicar una petición de escritura.

- *direcciónDeMemoria*, en formato hexadecimal.
- *datos*, cadena de 64 bytes, en este caso está recortada para poder mostrar la figura en el documento sin descuadrar. La dirección consta de 128 elementos (cada uno de estos es un hexadecimal, por lo que cada uno de estos elementos se representa con 4 bits, lo que hace 512 bits o 64 bytes).
- *idHilo*, en formato decimal indica el si se trata de peticiones a nivel de un único hilo o a mas de un hilo, este parámetro indica el hilo de destino.

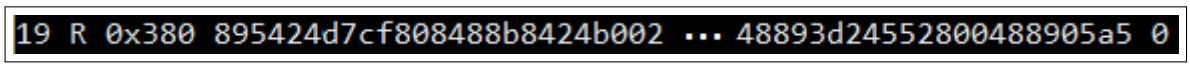

Figura 6.3: Línea del archivo de trazas

Para manejar estos datos NVMain emplea un objeto de tipo *TraceLine*, en el cual además de los cinco campos mencionados anteriormente también almacena un sexto campo que contiene los datos antiguos.

Capítulo 7

Estudio de NVMain

En este capítulo se explicará el funcionamiento del simulador empleando diagramas de diverso tipo para hacer más sencillo el entendimiento del flujo de ejecución, los distintos archivos y objetos que se emplean, así como una descripción en detalle de los diferentes controladores de memoria y de los métodos comunes y específicos del controlador de memoria anteriormente indicados.

7.1. El simulador

Antes de empezar este estudio es importante conocer el contenido del paquete NVMain tras la instalación, el cual que podemos ver en la figura 7.1 (No se han incluido el ejecutable ni archivos como el *README* o archivos para compilar el código).

El contenido de los diferentes subsistemas es el siguiente:

- *Banks*: Además del *BankFactory* contiene los dos tipos posibles de banks iniciales que se pueden utilizar *CachedDDR3Bank* y *DDR3Bank* (objeto que utiliza por defecto).
- *build*: Carpeta que se genera tras llevar a cabo la compilación y donde se encuentran los ejecutables que se lanzan desde el programa nvmain.fast.
- *Config*: Diferentes archivos de configuración de NVMain que se emplean para referenciar a los distintos tipos de memorias.
- *DataEncoders*: Junto con el *DataEncoderFactory* contiene la carpeta con el encoder que se emplea en nvmain.
- *Decoders*: Encontramos los archivos *Factory* y las carpetas referentes al decoder y el migrador de páginas.
- *Endurance*: Contiene los archivos *Factory* junto con las carpetas donde se encuentran los diferentes modelos de durabilidad.
- *FaultModels*: En su interior está el modelo de fallos.

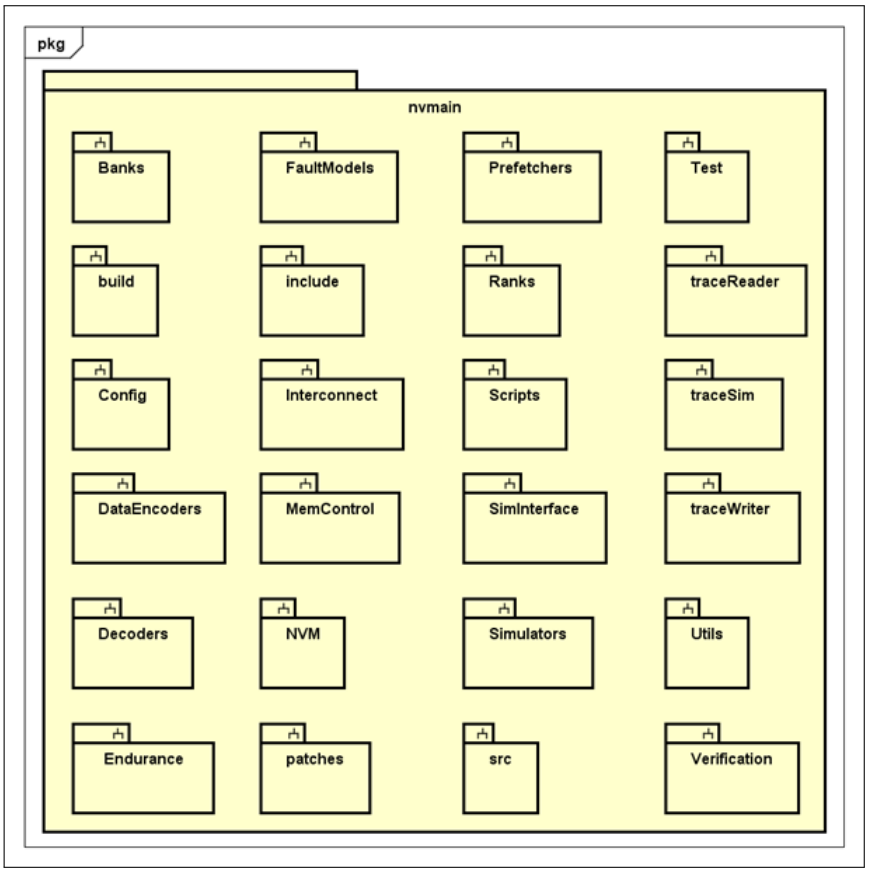

Figura 7.1: Contenido de NVMain

- *include*: Contiene diferentes objetos, que se emplean como contenedor de direcciones (*NVMAddres*), la definición del objeto que instancia las peticiones (*NVMainRequest*) y el objeto para almacenar los datos (*NVMDataBlock*) entre otros.

- *Interconnect*: Además de los archivos *Factory* contiene los dos tipos de interconnect disponibles *OffChipBus* y *OnChipBus*.

- *MemControl*: Contiene los archivos *Factory* y las carpetas correspondientes a los controladores de memoria que ya se han descrito anteriormente.

- *NVM*: En su interior aparece la clase superior del controlador de memoria, de la que cuelgan el resto de componentes.

- *patches*: Corresponde a los archivos necesarios para enlazar NVMain con otro simulador, en este caso *GEM5*.

- *Prefetchers*: Encontramos los archivos *Factory* y las carpetas con los archivos referentes a los Prefetcher.

- *Ranks*: Contiene los archivos *Factory* junto con la carpeta del objeto Rank que se crea.

- *Scripts*: Contiene un script en Python para recolectar los datos de la ejecución conjunta con *GEM5*.
- *SimInterface*: Interfaces para conectar NVMain con otros simuladores.
- *Simulators*: Contiene archivos para conectar con el simulador *GEM5*.
- *src*: Se encuentran las diferentes clases de las que heredan las clases concretas (ya sean del controlador de memoria, del interconnect, del rank... etc.) junto con clases con métodos que se implementan en diferentes objetos u objetos que se emplean para mantener datos o enviar información.
- *Tests*: En su interior podemos encontrar el archivo de trazas y un script Python para realizar pruebas.
- *traceReader*: Contiene los objetos necesarios para la lectura del archivo de trazas y su almacenamiento para poder tratarlo.
- *traceSim*: En su interior está la clase que inicia la simulación, el punto de entrada.
- *traceWriter*: Contiene los objetos que se emplean para mostrar durante la simulación las peticiones del archivo de trazas de manera detallada según se van sirviendo.
- *Utils*: En su interior se encuentran diversas clases referentes a múltiples funcionalidades del simulador (caché, *hook*, migrado, etc.).
- *Verification*: Contiene diferentes archivos de configuración para realizar test sobre el simulador.

7.2. Estructura de la memoria lógica

En esta sección se asociarán los elementos del archivo de configuración con la estructura de la memoria para explicar su arquitectura y como estos valores determinan la composición de la misma. Anteriormente ya se explicó de manera breve la estructura de la memoria ahora se indicará en detalle sus componentes.

En primer lugar, los valores del archivo de configuración empleados sobre los que se va a explicar la memoria son los referentes a la sección de código 7.2.

```
1  // Doble Data Rate
2  RATE 2
3
4  // Longitud del burst de datos
5  tBURST 4
6
7  // Número de banks por rank
8  BANKS 8
9
10 // Anchura del bus
```

```
11 BusWidth 64
12
13 // Bits que provee cada dispositivo
14 DeviceWidth 8
15
16 // Número de ranks por canal
17 RANKS 1
18
19 // Número de canales en el sistema
20 CHANNELS 4
21
22 // Número de filas en un bank
23 ROWS 8192
24
25 // Número de columnas visibles en un bank lógico
26 COLS 512
27
28 // Columnas del MAT (determina si hay o no paralelismo a nivel de subarray)
29 MATHeight 8192
```

En esta sección se hablará de dos conceptos que parecen similares pero que no lo son, por un lado se utilizara el concepto de bank el cual identifica tan solo al elemento bank almacenado en un device, figura 7.2 [1], y por el otro lado tenemos el bank lógico el cual identifica a todos los elementos correspondientes al mismo bank que están distribuidos entre los diferentes device de un rank, figura 7.3 [1]. En ambos ejemplos se ha dividido en dos banks por cada device con un total de 8 device en el rank.

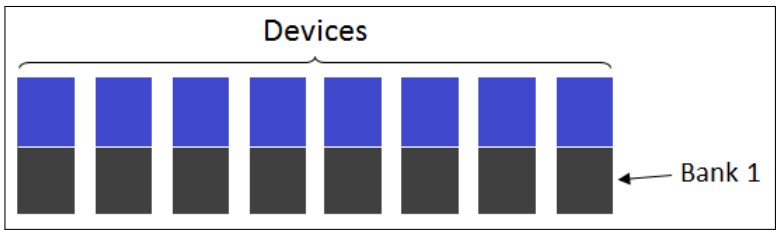

Figura 7.2: Concepto de bank.

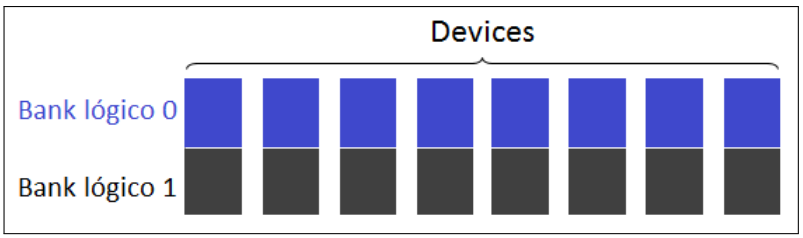

Figura 7.3: Concepto de bank lógico.

El elemento DIMM es el módulo de memoria de contactos duales (Dual In-line Memory Module), se corresponde con el objeto donde se encontraran todos los elementos que serán descritos a continuación, en la figura 7.4 [1] puede verse este elemento en solitario.

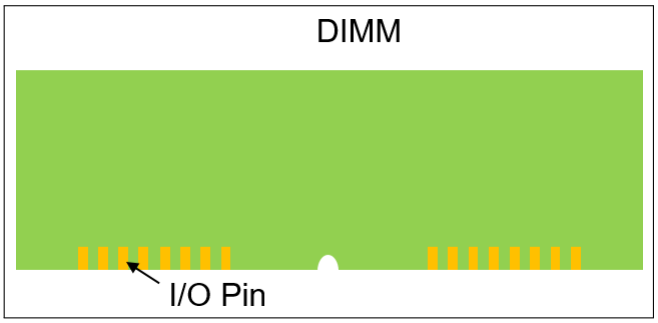

Figura 7.4: Objeto DIMM.

Además tenemos que tener en cuenta el número de canales que se utilizarán, ya que cada uno de estos canales que se añadan significará la duplicidad de los ranks y los elementos que estos contienen, es importante también indicar que cada canal estará gestionado por un controlador de memoria distinto, el parámetro correspondiente al número de canales se encuentra en la sección de código 7.2. El número de ranks que encontraremos por cada canal se indica por el parámetro asociado, este se puede encontrar en la sección de código 7.2, este elemento rank puede observarse en la figura 7.5 [1] junto con el DIMM y los device asociados.

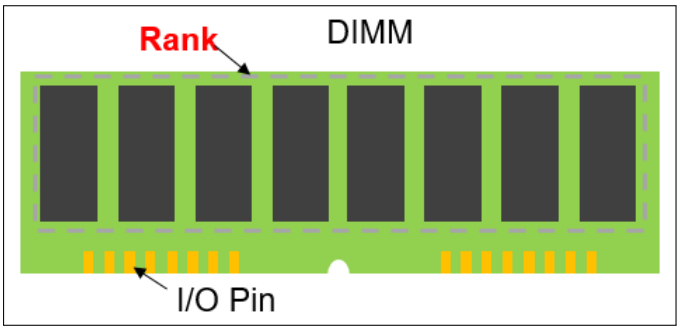

Figura 7.5: Objeto DIMM junto con el rank y sus device asociados.

El número de devices que encontraremos en cada rank viene determinado por la fórmula 7.1. Los valores correspondientes para poder resolver la fórmula se encuentran en la sección de código 7.2.

$$\text{Número de dispositivos} = \text{BusWidth} / \text{DeviceWidth} \tag{7.1}$$

Los device que encontramos en un DIMM contienen en total 8 banks lógicos, los cuales están presentes en cada uno de los device del DIMM, por lo que a la hora de acceder a una fila de un bank se accede en todos estos device al bank concreto de la petición. El parámetro que indica los banks presentes en cada device se puede ver en la sección de código 7.2, en la figura 7.6 [1] puede observarse los banks que contiene un device.

Cada subarray está formado por 8 MATs, esto se obtiene de la fórmula 7.2. De cada subarray se obtiene un bit en el ciclo de subida y otro bit en el ciclo de bajada (DDR). Los parámetros del archivo de configuración referentes a estos datos se pueden ver en la sección de código 7.2. En la figura 7.7 [1] se muestra el número de MATs que forman un subarray y como cada bank

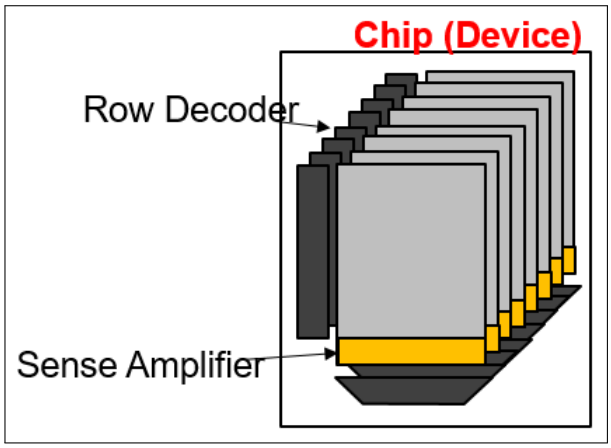

Figura 7.6: Banks que contiene un device.

solo contiene un subarray.

$$\text{Número de MATs} = \text{tBURST} * \text{DDR} \tag{7.2}$$

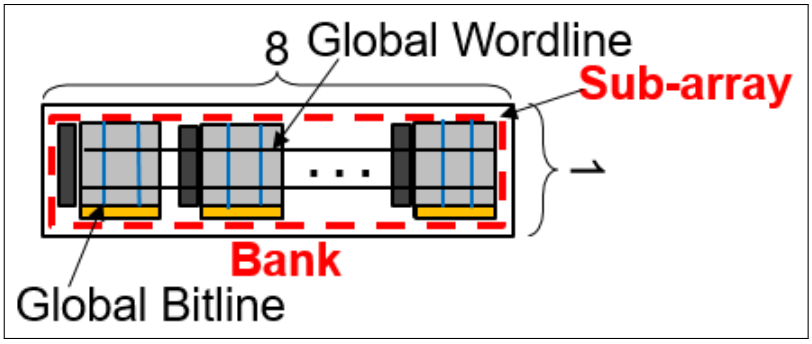

Figura 7.7: Subarray contenido en un bank.

Por lo descrito anteriormente sabemos que como el ancho del bus es de 64 bits, en cada ciclo se proveerán 128 bits (DDR) y en cuatro ciclos se proveerán 512 bits (Burst de datos). Esto corresponde al tamaño de la fila del buffer, tamaño de página o row buffer, la manera de calcular este valor es mediante la fórmula 7.3. Estos 512 bits se proveen a través de un dispositivo, por lo que al disponer de 8 dispositivos el total son 512 bytes.

$$\begin{aligned}\text{Fila del buffer} = {} & \text{Bits por dispositivo} * \text{Dispositivos por rank} * \text{Número de ranks} \\ & * \text{Data rate} * \text{Burst de datos}\end{aligned} \tag{7.3}$$

Si queremos calcular el tamaño de la memoria que se crea para la simulación con NVMain tan solo se deben aplicar las fórmulas 7.4, 7.5 y 7.6.

$$\text{Tamaño total del canal} = \text{Tamaño del bank lógico} * BANKS * RANKS \tag{7.4}$$

$$\text{Tamaño del bank lógico} = ROWS * COLS * \text{Tamaño de palabra (en bytes)} \tag{7.5}$$

$$\text{Tamaño de palabra (en bytes)} = \frac{DeviceWidth * \text{burst de datos} * \text{Data rate} * \text{num. dispositivos}}{8\frac{\text{bits}}{\text{bytes}}} \tag{7.6}$$

Aplicando las fórmulas 7.4, 7.5 y 7.6 obtenemos que la memoria empleada dispone de cuatro canales con un gigabyte de capacidad por canal (4 gigabytes totales), cada canal tiene un rank, y estos ranks tienen asociados 8 banks lógicos con 125 MB (megabytes) cada uno. Además, cada uno de estos banks lógicos está formado por 1 solo subarray (ya que el parámetro *MATHeight* es igual a *ROWS*), por lo que no hay paralelismo a nivel de subarray.

Por lo tanto podemos ir desgranando cada elemento para obtener el tamaño individual de cada componente de la siguiente manera:

- 1 rank almacena 1 GB y en este rank encontramos 8 dispositivos.
- 1 dispositivo almacena 1GB/8 y contiene 8 banks.
- 1 bank almacena 1/64 GB y contiene tan solo 1 subarray.
- 1 subarray almacena 1GB/64 y contiene 8 MATs.
- 1 MAT almacena 1GB/512, lo que son 2^{21} bytes o 2^{24} bits.
- Las filas de un MAT están determinadas por *MATHeight*, 2^{13}, por lo que el número de columnas de un MAT son 2^{11}.

Se pueden observar los datos anteriores referidos al tamaño de un MAT en la figura 7.8.

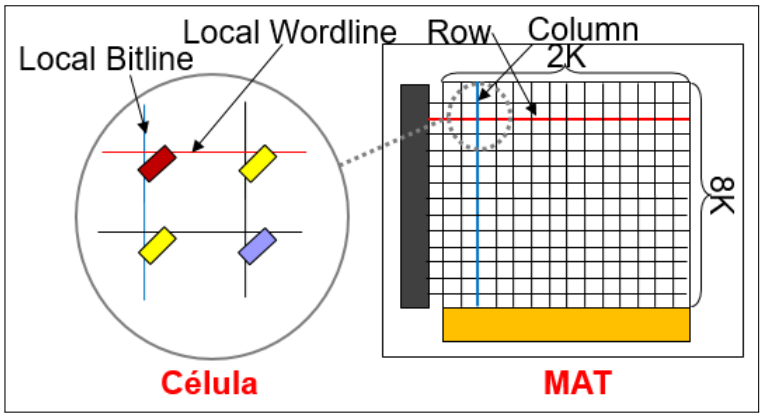

Figura 7.8: Arquitectura del objeto MAT.

7.3. Inicio de la simulación

En esta sección se va a describir el funcionamiento de NVMain en el momento que se inicia la simulación. Un ejemplo de ejecución de NVMain es el siguiente:

```
./nvmain.fast Config/ArchivoConfig.config Test/Traces/ArchivoTrazas.nvt 0
```

Donde 0 es el número de ciclos que debe ejecutar la simulación, si indicamos 0, como en este caso, se llevará a cabo la simulación completa del archivo de trazas, mientras que si se indica otro número de ciclos mayor que cero la simulación se detendrá tras alcanzar ese número de ciclos. Es importante conocer que estos ciclos que se indican son ciclos de memoria, los ciclos de CPU que se ejecutarán serán más debido a que tiene una mayor frecuencia (es más rápida).

El programa principal que inicia la simulación puede ser encontrado en *nvmain/traceSim/traceMain.cpp* en el cual se encuentra el método principal del programa, en este, se crea un objeto *TraceMain* con el que se ejecuta el método *RunTrace*. En este método podemos observar el bucle principal de la simulación (bucle *while* en el que se permanece hasta que se excede el número de ciclos indicado o se completa la simulación).

Pero antes de llegar a esa parte se procederá a describirlo en detalle. En primer lugar en este método se crean varios objetos de tipo:

- *Stats*, donde se almacenarán las estadísticas de la ejecución.
- *Config*, donde estarán contenidas la configuración concreta de los elementos de NVMain.
- *GenericTraceReader*, objeto para leer el archivo de trazas empleando el objete *TraceLine*.
- *TraceLine*, objeto para almacenar las peticiones que se van leyendo del archivo de trazas.
- *SimInterface*, en este caso *NullInterface* ya que no se emplea *GEM5* al iniciar la simulación desde NVMain.
- *NVMain*, objeto hijo al que se mandaran las peticiones
- *EventQueue*, objeto referente a la cola de eventos principal.
- *GlobalEventQueue*, objeto referente a la cola de eventos global (se emplea para configurar el subsistema de memoria, indicar la frecuencia de funcionamiento de la CPU y el avance de ciclos).
- *TagGenerator*, variable que referencia al objeto generador de etiquetas.

Después de haber creado estos objetos lleva a cabo la comprobación del número de argumentos es el correcto, 4 (ejecutable, archivo de configuración, archivo de trazas y número de ciclos). A continuación, se llama al método *Read* que se encuentra en *src/Config.cpp*, el cual se encarga de recoger los elementos del archivo de configuración seleccionado, almacenando todos los parámetros que se encuentran en el archivo junto con sus valores.

Lleva a cabo diferentes asignaciones, como son la del objeto de configuración a la interfaz de simulación que se está empleando, el objeto *NVMobject* la *mainEventQueue* que se ha creado con la *eventQueue* que maneja este objeto y el objeto *NVMobject* a la *globalEventQueue* que se ha creado con la que maneja este objeto.

Asigna en el objeto *NVMobject* el objeto *tagGenerator* que se ha creado a *tagGen* que es el que maneja este objeto, comprueba si en la línea de comandos se han añadido parámetros que sustituirán a los indicados en el archivo de configuración (en cuyo caso los reemplaza) y si se ha indicado un archivo para almacenar las estadísticas lo abre para ese propósito.

Crea los *hooks* para poder fisgar las peticiones, añade como hijo a *nvmain* y establece también que su padre es este método *main* (Inicio de la jerarquía de árbol), asigna la frecuencia de la CPU en el objeto *EventQueue* referente a la variable *globalEventQueue* (frecuencia del sistema principal), mediante *AddSystem* establece la frecuencia a la que funcionará *nvmain* (el subsistema de memoria) y asigna en el objeto *src/SimInterface* la configuración que se ha generado al parámetro que maneja este objeto.

Se hace una llamada al método *SetConfig* de *nvmain/NVM/nvmain.cpp* e imprime la jerarquía, selecciona el lector de trazas, crea el objeto y carga el número de ciclos que se realizarán (si el valor es 0 se realizará una simulación completa), tras esto establece el ciclo actual como 0 e inicia el bucle principal, que termina cuando se alcanzan los ciclos objetivo o se completa la simulación, la lógica de este bucle puede observarse en la figura 7.9.

A continuación se describe el flujo de este bucle en detalle. En este bucle, se comprueba si es posible obtener el siguiente elemento de la traza empleando para ello el método *GetNextAccess* de *NVMainTraceReader*. El cual se encarga de leer las líneas del archivo y cargar sus valores en variables comprobando que estos valores son correctos, si no puede leer la siguiente línea fuerza el drenaje de las colas de peticiones, espera a que termine el drenaje, incrementa el ciclo y reintenta el drenaje si ha fallado anteriormente antes de acabar el bucle. En caso de poder leer la línea del archivo de trazas, genera un objeto petición de tipo *NVMainRequest*, en el cual almacena los datos obtenidos del archivo de trazas, además lo marca como incompleto e indica su propietario. Comprueba si debe o no tener en cuenta los ciclos que aparecen en el archivo de traza, si es así sustituye su valor en el registro por 0.

Tras esto comprueba si la operación es reconocida por el simulador (operación de lectura, read, u operación de escritura, write). En caso que la petición ocurre después de los ciclos indicados se termina el bucle. Comprueba si el ciclo en el que se debe realizar la petición es posterior al ciclo actual, en cuyo caso, espera a que pasen los ciclos necesarios antes de continuar, después comprueba si se ha sobrepasado el límite de ciclos a simular para salir del bucle en ese caso. Mientras el objeto hijo (objeto *nvmain*), no esté disponible para la petición (*IsIssuable*):

- Comprueba si se ha sobrepasado el límite de ciclos.
- Avanza 1 ciclo.
- Recupera el ciclo actual.
- Vuelve a comprobar si está disponible.

Una vez está listo, incrementa *outstandingrequest* (contador con la peticiones pendientes), manda esta petición a la cola de comandos mediante *IssueCommand*, comprueba de nuevo si el ciclo actual ha superado la marca y vuelve al principio del bucle.

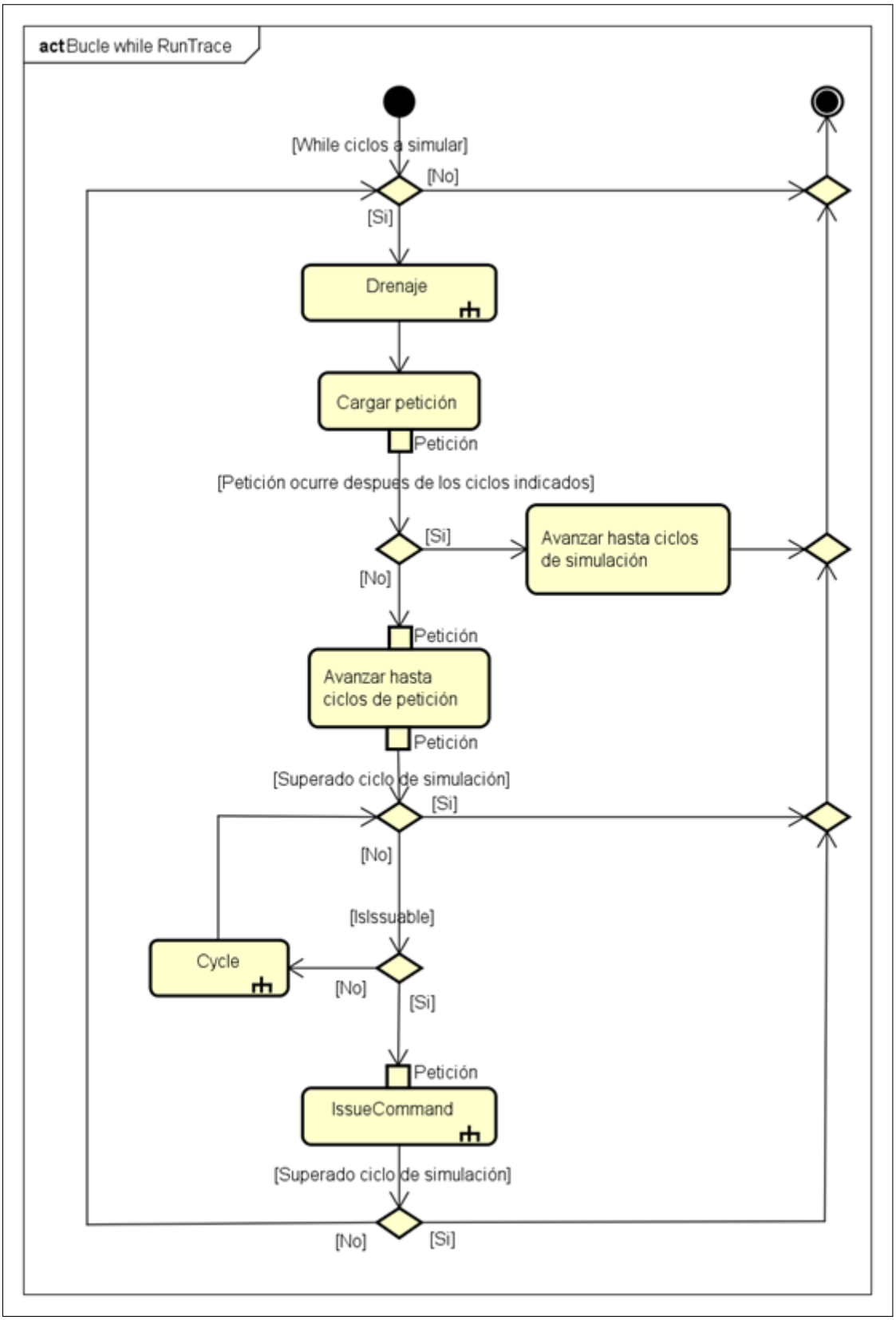

Figura 7.9: Bucle principal del programa

Anteriormente se ha indicado que desde este bucle principal se emplean dos métodos, uno para comprobar si el objeto hijo está disponible para la petición y otro para enviar la petición a la cola de comandos.

En la figura 7.10 se muestra lo que ocurre al realizar la operación de *IsIssuable*. En el cual se comprueba si el controlador de memoria puede aceptar la petición. Se ha empleado un diagrama de secuencia para mostrar las diferentes llamadas, los componentes que intervienen, los archivos y sus directorios.

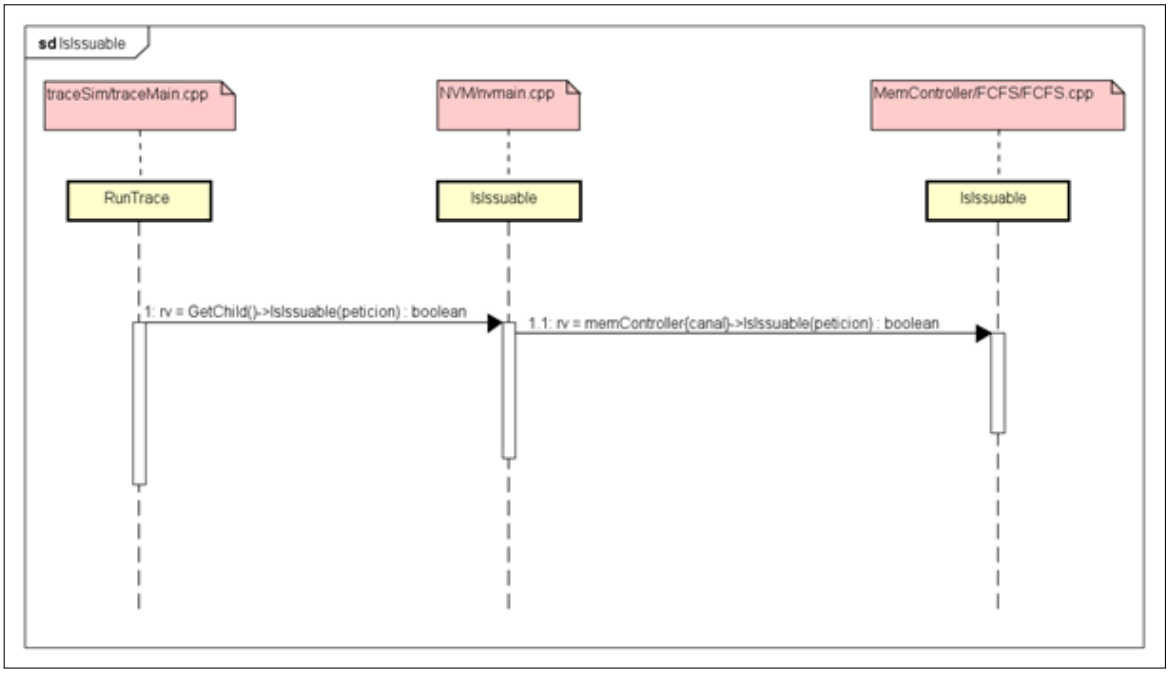

Figura 7.10: Ejecución de IsIssuable

En el diagrama de secuencia de la figura 7.11 se muestra el funcionamiento de *IssueCommand*, en el cual se inserta la petición a la cola de comandos, las diferentes llamadas que se realizan, los componentes que intervienen, los archivos y los directorios.

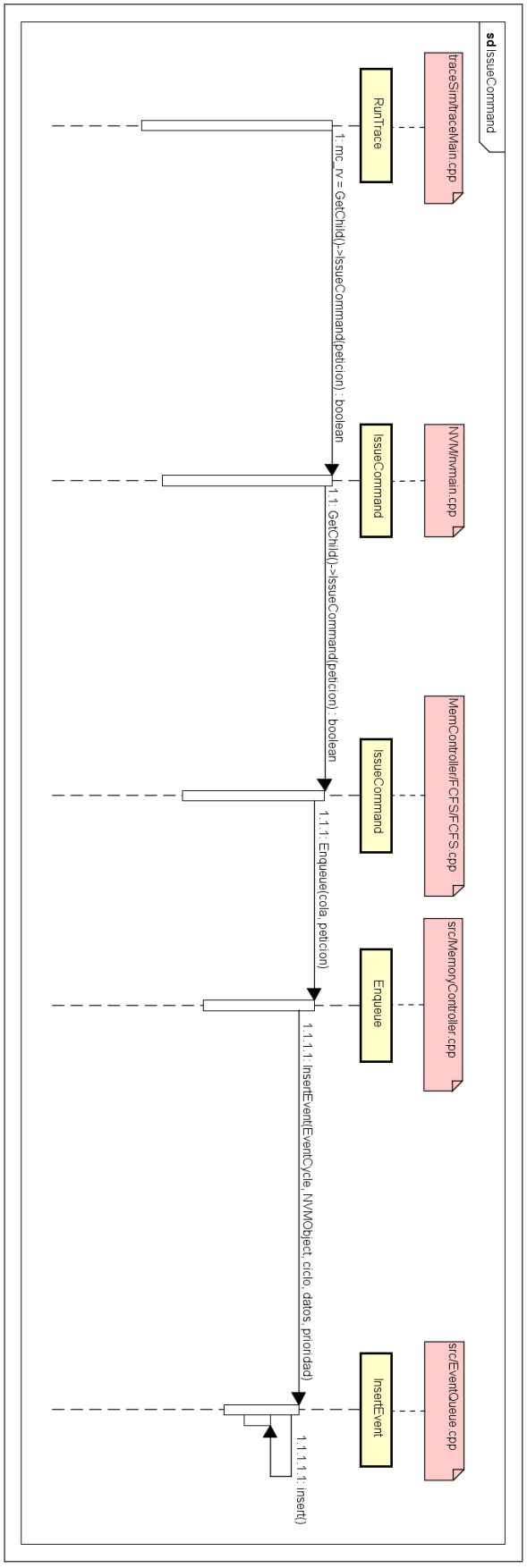

Figura 7.11: Ejecución de IssueCommand

7.4. Controladores de memoria en detalle

En esta sección se va a llevar a cabo una descripción en detalle de los controladores de memoria, para poder comprender su funcionamiento, las acciones realizadas en sus diferentes métodos y que otros objetos se emplean.

Todos los controladores de memoria heredan del objeto *MemoryController.cpp* que podemos encontrar en el directorio *nvmain/src*. Aunque estos controladores implementan sus métodos propios, al finalizar estos métodos llaman al método genérico del objeto que heredan.

Desde el objeto hijo del método principal que inicia la simulación, *nvmain*, se emplea *MemoryControllerFactory* para crear el objeto del controlador de memoria, creando un controlador de memoria para cada canal.

7.4.1. FCFS

En este controlador de memoria las páginas se cierran tras cada lectura/escritura, en cada petición añade una activación antes de la lectura/escritura y añade una precarga. Todos los banks y ranks están en modo activo. No lleva a cabo una gestión de la energía. Se basa en resolver las peticiones como si de una cola *FIFO* (First In First Out, primera en llegar primera en ser servida) se tratara. En este controlador encontramos los siguientes métodos:

1. *SetConfig*. Cuando se crea el objeto se le da un valor inicial al tamaño de la cola, pero se comprueba si desde el archivo de configuración (*.config*) se ha indicado este tamaño (campo *QueueSize*), en cuyo caso se carga. Se utiliza un objeto de tipo *Params.cpp* que sirve para cargar los valores del archivo de configuración referentes a la memoria.

   En el caso que *createChildren* sea cierto, se crea el traductor de direcciones (objeto *AddressTranslator*). Se cargan los valores referentes a las variables cols, rows, banks... etc, en el caso de rows y subarrays se comprueba si se ha establecido un valor a *MATHeight*, en caso negativo se establece un solo subarray con un gran número de filas, en caso de que si tenga un valor establecido se emplea la fórmula 7.7 para calcular el número de subarrays, los valores de ambas variables se indican en el archivo de configuración.

   $$\text{Número de subarrays} = \text{ROWS} \ / \ \text{MATHeight} \tag{7.7}$$

   Una vez se han poblado estas variables crea un objeto de tipo *TranslationMethod* y lo inicializa, este objeto es utilizado junto con el traductor de direcciones para llevar a cabo el trabajo del decoder. A continuación, crea un interconnect y lo inicializa. Por último configura el controlador de memoria con los valores y objetos descritos anteriormente empleando el método *SetMappingScheme()*.

   Se establece el número de colas de comandos, por defecto se asume que estas colas serán a nivel de bank, pero esto puede modificarse habiendo tres posibilidades:

   a) A nivel de bank.

b) A nivel de rank.

c) A nivel de subarray.

Se crean estas colas y se inicializan, para finalizar este método lleva a cabo un refresco de las memorias si es necesario.

2. *RegisterStats*. Llama a la clase *Stats.cpp* pasándole el indicador que se almacenara como estadística, tras esto llama a la clase *MemoryController* para guardar estadísticas referentes a los ciclos de la simulación y los *wake ups* (ciclos en lo que se despierta al controlador para realizar comprobaciones de la memoria).

3. *RequestComplete*. En el caso de tratarse de una petición de:

   a) Lectura.

   b) Precarga de lectura.

   c) Escritura.

   d) Precarga de escritura.

   Se cambia el estado a completado, se obtiene el ciclo actual, se actualizan los tiempos de latencias y se llama al método del *MemoryController*. En este método, se comprueba si se es el dueño de la petición, en cuyo caso se elimina de la cola y se termina, en caso de no ser así, se le pasa al padre la petición para que haga la comprobación anterior.

4. *Cycle*. En primer lugar se busca la petición más antigua que está preparada mediante *FindOldestReadyRequest*. En caso de no encontrar ninguna, se busca una petición dirigida a un bank que no esté activo. En el caso de tener éxito uno de estos métodos, devolverán la petición encontrada para poder realizar la operación correspondiente sobre la memoria principal mediante el método *IssueMemoryCommands*. Tras esto se lleva a cabo la ejecución de los elementos de la cola de comandos si es posible mediante *CycleCommandQueues* y se llama al método Cycle de *MemoryController.cpp*. A continuación se describen estos métodos más en profundidad:

   a) *FindOldestReadyRequest*: recorre mediante un iterador la cola de transacciones de inicio a fin, se obtiene el identificador de la cola, con ese identificador se comprueba si la cola de comandos está vacía, si no es así se continúa con la siguiente interacción comprobando la siguiente petición.

   Se recupera el bank y el rank al que pertenece la petición y se busca la primera petición de la cola (*FIFO*, las más antiguas se devuelven primero) para la cual: el bank de destino esté activo, el bank no esté esperando por un refresco, no se va a interrumpir el refresco de cola, la cola de comandos no esté vacía, el ciclo de llegada de la petición sea diferente del ciclo actual y el predicado sea true. En ese caso se almacena la petición y se elimina de la cola de transacciones.

Se comprueba si es la última petición a la fila del buffer, en cuyo caso se marca como tal empleando el operador binario |. Se marca como true la variable de control, se termina la iteración del bucle y se devuelve la variable de control.

b) *FindClosedBankRequest*: muy similar al anterior, salvo que las condiciones para encontrar la petición en la cola son distintas. Se comprueba que: el bank no esté inactivo, el bank no esté esperando un refresco, no se va a interrumpir el refresco de cola, la cola de comandos está vacía, el ciclo de llegada de la petición es distinto del ciclo actual y el predicado sea true.

c) *IssueMemoryCommands*: se recupera la columna, la fila, el bank, el rank y el subarray referentes a la petición, además se obtiene una referencia al subarray de destino, el subconjunto de los elementos cargados en los sense amplifiers y el identificador de la cola de comandos. Tras esto trata de lanzar la petición por métodos alternativos (caching [8]). Realiza diferentes comprobaciones para ver si es necesario llevar a cabo alguna activación (ya sea de la cola del bank y rank que referencia la petición, el subarray o el conjunto de de la fila del buffer que se encuentra en los sense amplifiers) en cuyo caso se lanza la activación y se resetea el contador que referencia a la inanición (empleado para cambiar entre escritura y lectura en función del tipo de controlador de memoria). Si no hay que llevara a cabo ninguna activación se incrementa el contador de inanición y añade la petición a la cola de comandos, pero se dan dos casos:

1) Es necesario realizar una precarga.
2) No es necesaria la precarga.

Si no se da ninguno de los casos marca la variable de control como false (por lo que no programa ningún comando de *wake up*) y devuelve esta variable de control.

d) *CycleCommandQueues*: en primer lugar comprueba si el ciclo corresponde con un refresco, en cuyo caso ya no es necesario hacer nada más, en caso contrario, se recorren las colas de comandos.

En el caso de que la cola de comandos no esté vacía, el último ciclo de emisión sea diferente del ciclo actual y el primer elemento de la cola de comandos pueda ser emitido, lo que hace es, se marca como emitido, se realiza un refresco de la cola si es necesario y el último ciclo de emisión toma el valor del ciclo actual. Se elimina el elemento de la cola de comandos (*CleanupCallback*) y se inserta en la cola de eventos, si no fue posible llevar a cabo esta tarea se realiza mediante el método *InsertCallback* (*EventQueue*).

Tras esto, se comprueba si para la siguiente iteración la cola de comandos (también llamada cola de bank) hay una transacción disponible, se establece el siguiente ciclo como *wake up* del sistema y se inserta en la cola de eventos este *wake up*. Se incrementa el valor de la cola actual y se termina.

En el caso que la cola de comandos no esté vacía y no se den las condiciones adicionales indicadas anteriormente. Se muestra el mensaje de aviso indicando que no es posible

lanzar la ejecución durante el ciclo actual y que no será posible hasta dentro de un largo periodo.

e) *Cycle* (*MemoryController.cpp*): comprueba si ya se va a lanzar otro evento de la cola de eventos durante ese ciclo, en caso afirmativo termina. Si no es así, comprueba si la cola de comandos está vacío (o lo estará en el siguiente ciclo), y hay una transacción disponible en la cola de transacciones, en cuyo caso la inserta a la cola de eventos y termina.

7.4.2. FRFCFS

Este controlador es similar al anterior, también funciona como una cola *FIFO*, pero con una particularidad, prioriza las peticiones que están dirigidas a la misma fila del buffer pero teniendo en cuenta el número de peticiones que ya se han lanzado llevando a cabo esta política para evitar la inanición del resto de las peticiones, esto se controla mediante un número máximo de peticiones a la fila del buffer mediante el parámetro *StarvationThreshold*. A continuación se detallan las partes que se diferencian con respecto al controlador anterior:

1. *SetConfig*. Comprueba si en el fichero de configuración se ha indicado el valor para *starvationThreshold* en cuyo caso establece el valor de la variable con el valor rescatado del archivo de configuración (valor por defecto al inicializar 4).

2. *IsIssuable*. Aunque es igual al método del controlador anterior para acceder a la cola de transacciones emplea un puntero, y de la misma manera que antes solo hay una cola.

3. *RequestComplete*. Al comenzar lleva a cabo una comprobación con las peticiones de escritura y precargas de escritura, si estas han sido canceladas (comprobación empleando el flag), se vuelven a colocar al principio de la cola como si no hubiera pasado nada y termina. En el caso que no se haya encontrado el problema descrito continua de manera normal realizando las mismas operaciones que en el controlador anterior.

4. *Cycle*. En este caso el método realiza un mayor número de operaciones que en el controlador anterior. En primer lugar se buscan peticiones que sufran de inanición empleando *FindStarvedRequest*, si no encuentra ninguna busca peticiones a la misma fila del buffer que ya está cargado en los Sense Amplifiers mediante *FindRowBufferHit*, en caso de no encontrar ninguna comprueba si es posible acceder a través de la caché (acceso sin emplear un ciclo normal de activación) mediante el método *FindCachedAddress*, en caso de no encontrar ninguna busca peticiones de escrituras que hayan sido pausadas para lanzar una petición de lectura mediante *FindWriteStalledRead*, en caso de no encontrar ninguna las operaciones a realizar son las ya expuestas en el controlador anterior utilizando los métodos *FindOldestReadyRequest*, *FindClosedBankRequest*, *IssueMemoryCommands*, *CycleCommandQueues* y la llamada a la clase *Cycle* de *MemoryController* del cual hereda. A continuación se describen estos nuevos métodos que se acaban de mencionar:

a) *FindStarvedRequest*: recorre mediante un iterador la cola de transacciones de inicio a fin, se obtiene el identificador de la cola, con ese identificador se comprueba si la cola de comandos está vacía, si no es así se continúa con la siguiente iteración comprobando la siguiente petición.

Se recupera el bank, el rank, la fila, la columna y el subarray a los que pertenece la petición y se busca la primera petición para la cual se cumple que:

- La cola referente al bank y rank destino esté activada.
- El subarray no esté activado o la fila efectiva no coincida con la fila cargada en los sense amplifiers o el subconjunto de la fila del buffer que se encuentra en los sense amplifiers no coincide con el necesario.
- El bank no está esperando por un refresco.
- No se va a realizar un refresco de la cola referente al bank y rank destino esté activada.
- Se ha alcanzado el máximo de peticiones a la misma fila del buffer.
- El ciclo de llegada es diferente al ciclo actual.
- La cola de comandos está vacía.
- El predicado es True.

En caso de cumplir las condiciones anteriormente mencionadas, se almacena la petición para servirla y se elimina de la cola de transacciones. Por último se comprueba si es la última petición en cuyo caso se marca como tal empleando el operador binario |=. Se marca como true la variable de control y se finaliza el bucle y se devuelve la variable de control.

b) *FindRowBufferHit*: recorre mediante un iterador la cola de transacciones de inicio a fin, se obtiene el identificador de la cola, con ese identificador se comprueba si la cola de comandos está vacía, si no es así se continúa con la siguiente iteración comprobando la siguiente petición.

Se recupera el bank, el rank, la fila, la columna y el subarray a los que pertenece la petición y se busca la primera petición para la cual se cumple que:

- La cola referente al bank y rank destino esté activada.
- El subarray está activo.
- La fila efectiva coincide con la fila cargada en los sense amplifiers.
- El subconjunto de la fila del buffer que se encuentra en los sense amplifiers coincide con el necesario.
- El bank no está esperando por un refresco.
- No se va a realizar un refresco de la cola referente al bank y rank destino esté activada.
- El ciclo de llegada es diferente al ciclo actual.
- La cola de comandos está vacía.

- El predicado es true.

En caso de cumplir las condiciones anteriormente mencionadas, se almacena la petición para servirla y se elimina de la cola de transacciones. Por último se comprueba si es la última petición en cuyo caso se marca como tal empleando el operador binario |=. Se marca como true la variable de control y se finaliza el bucle y se devuelve la variable de control.

c) *FindCachedAddress*: recorre mediante un iterador la cola de transacciones de inicio a fin, se obtiene el identificador de la cola, se transforma la petición en una petición de tipo caché mediante el método *MakeCachedRequest*, tras esto se comprueba que:

- La cola de comandos no este vacía.
- Se comprueba si el hijo es utilizable empleando como argumento la petición de tipo caché (método *IsIssuable* con el cual se desciende por la estructura en árbol: interconnect, rank, bank, subarray).
- El ciclo de llegada es diferente al ciclo actual.
- El predicado es true.

En caso de cumplir las condiciones anteriormente mencionadas, se almacena la petición para servirla, se elimina de la cola de transacciones, se elimina la petición de tipo caché que se había creado para liberar espacio de la memoria se marca la variable de control como true y se devuelve.

d) *FindWriteStalledRead*: en primer lugar se comprueba si se ha establecido como true el parámetro *WritePausing* el cual sirve para saber si se pueden pausar las escrituras para llevar a cabo lecturas, en caso de ser así se devuelve false, pues no se va a encontrar ninguna petición de escritura pausada para realizar una escritura. Tras esto se recorre mediante un iterador la cola de transacciones, si la petición a la cual apunta el iterador no es una petición de escritura se salta a la siguiente iteración, se comprueba que la cola de comandos no esté llena y se cargan los valores del bank, rank, la fila y la columna y el subarray referentes a la petición. Se obtiene una referencia al subarray destino, en caso de no poder obtener esta referencia se termina devolviendo false. Se transforma la petición en una petición de activación y se comprueba que:

- El bank destino no está esperando un refresco.
- La cola referente al bank y rank destino esté activada.
- Se está ejecutando una escritura.
- Se comprueba si el hijo es utilizable empleando como argumento la petición de entrada al método o con la petición de tipo activación (empleando el método *IsIssuable* con el cual se desciende por la estructura en árbol: interconnect, rank, bank, subarray).
- El ciclo de llegada es diferente al ciclo actual.
- La cola de comandos está vacía.
- El predicado es true.

En caso de cumplir las condiciones anteriormente mencionadas, se comprueba si no se encuentra en el último ciclo de escritura y el parámetro *pauseMode* esta establecido como *PauseMode_Normal*, en cuyo caso se elimina la petición de activación que se había generado, se finaliza el bucle y se devuelve true. Si no se cumple la condición anterior, se almacena la petición para servirla, se elimina de la cola de transacciones y se elimina la petición de activación creada. Por último se comprueba si es la última petición en cuyo caso se marca como tal empleando el operador binario |=. Se finaliza y se devuelve true.

7.4.3. FR-FCFS with Write-Buffering

Este controlador añade un nivel más de complejidad al anterior, ya que emplea dos colas para las peticiones (colas de transacciones) en lugar de solo una como los controladores vistos hasta ahora. Encontramos una cola de escritura y una cola de lectura, para gestionar estas dos colas emplea dos parámetros que marcan la cota superior, *HighWaterMark*, cota que al ser alcanzada inicia el drenaje de la cola de escritura, y una cota inferior, *LowWaterMark*, cota que al ser alcanzada finaliza el drenaje de la cola de escritura. Aunque emplea los mismos métodos tiene sus propias características, por ello, se describirán las partes que sean diferentes:

1. *SetConfig*. Se carga del archivo de configuración el tamaño de ambas colas en las variables *ReadQueueSize* y *WriteQueueSize*, también almacena los valores referentes a las cotas inferior y superior y comprueba si la cota superior es mayor al tamaño de la cola de escritura o si la cota inferior es mayor que la cota inferior, en cuyo caso lanza un aviso por pantalla.

2. *IsIssuable*. Lleva a cabo una comprobación en función del tipo de petición entrante con respecto a su cola correspondiente indicando que no es utilizable si la cola no tiene espacio libre, en caso de tratarse de una petición de escritura esta no va a poder lanzarse si se está en proceso de drenaje, hasta que el drenaje no finalice no será incluida en la cola.

3. *IssueCommand*. Comprueba en primer lugar si es utilizable (*IsIssuable*, tras esto se establece el ciclo de llegada como el ciclo actual y se añade a la cola correspondiente).

4. *Cycle*. En esté método podemos observar dos comportamientos muy diferenciados, cuando se está ejecutando el drenaje (se actúa sobre la cola de escritura) y cuando no se está ejecutando el drenaje (se actúa sobre la cola de lectura), para alternar entre ambos comportamientos lleva a cabo dos comprobaciones al iniciar el método:

   a) Paso de la cola de lectura a la cola de escritura. Para saber si se debe realizar esta transición, se comprueba si:

   - No se está llevando a cabo el drenaje.
   - El número de peticiones en la cola de escritura es igual o mayor a la cota superior.
   - No se está forzando el drenaje.

En cuyo caso, se almacena el ciclo de inicio del drenaje, el número de elementos en la cola de lectura y se indica que se inicia el drenaje estableciendo la variable *m_ draining* a true.

b) Paso de la cola de escritura a la cola de lectura. Para saber si se debe realizar esta transición, se comprueba si:

- Se está llevando a cabo el drenaje.
- El número de peticiones en la cola de escritura es igual o menor a la cota inferior.
- No se está forzando el drenaje.

En cuyo caso, se almacena el ciclo de fin de drenaje y se almacenan un gran número de estadísticas en caso que este último ciclo de drenaje no coincida con el cero:

- El número de peticiones de lectura añadidas a la cola de lectura durante el drenaje.
- El tamaño máximo y mínimo alcanzado en la cola de lectura durante el drenaje.
- El tamaño de la cola de lectura antes del drenaje en función de si supone un valor mínimo o máximo global.
- El número de peticiones de entrada a la cola de lectura durante el drenaje en función de si supone un valor mínimo o máximo global.
- El número total de ciclos de drenaje que ha habido.
- El número total de ciclos de drenaje en función de si supone un valor mínimo o máximo global.
- Se incrementa el total de drenajes realizados y el número de peticiones servidas durante el drenaje.
- Las peticiones de escritura durante un drenaje en función de si supone un valor mínimo o máximo global.
- El intervalo entre dos drenajes.
- El intervalo entre dos drenajes en función de si supone un valor mínimo o máximo global.
- El total de ciclos de lectura realizados durante la anterior iteración con la cola de lectura.
- El valor anterior en función de si supone un valor mínimo o máximo global.

Una vez establecido el valor de esas variables para poder almacenarlo, se resetea la variable correspondientes a las peticiones por drenaje y cambia el valor del último ciclo de drenaje y se modifica la variable *m_ draining* estableciendo su valor a false para cambiar a la cola de lectura.

En este momento, se genera un objeto de tipo *NVMainRequest* y lo emplea para llevar a cabo el procesamiento de estas peticiones de la cola de lectura o escritura en función del estado en el que se encuentre, para distinguir esto se comprueba si se está drenando o si se está forzando el drenaje y además el tamaño de la cola de lectura es cero:

- Cola de escritura. Se llevan a cabo la ejecución de los métodos *FindStarvedRequest*, *FindRowBufferHit*, *FindCachedAddress*, *FindOldestReadyRequest* y *FindClosedBankRequest*, los cuales ya han sido explicados en el anterior controlador.
- Cola de lectura. Se lanzan los mismos métodos que antes para la cola de lectura, la diferencia entre ambos métodos son las variables de estadísticas empleadas.

Finalmente se lanzan *IssueMemoryCommands*, *CycleCommandQueues* y el método *Cycle* correspondiente al objeto *MemoryController*, los cuales ya han sido explicados.

5. *Drain*. Cambia el forzado de drenaje a true y devuelve true.

7.4.4. PerfectMemory

Como ya se indicó antes, en este controlador no encontramos latencias, además este controlador de memoria es considerado como un nodo final del sistema, es decir, las peticiones no van más allá. Encontramos los métodos:

- *IssueCommand*. Que realiza la inserción de la petición en la cola de comandos mediante *InsertEvent* y se devuelve true. No se lleva a cabo ninguna comprobación sobre si la cola está o no llena, ya que se supone que este controlador es capaz de servir las peticiones inmediatamente.
- *Cycle*. No lleva a cabo ninguna operación, ni llama al método del padre *MemoryController*.
- *InsertEvent*. Mismo cosa que con el método anterior.

7.4.5. Parámetros de salida

La salida de estos controladores de memoria nos devuelve una gran cantidad de parámetros, los cuales están asociados a diferentes componentes del simulador o son de carácter general, de manera común para todos los controladores en el simulador nos encontramos parámetros que referencian:

1. Al subarray:
    - *subArrayEnergy*, muestra en nanojulios la energía consumida por el subarray.
    - *activeEnergy*, muestra la energía en nanojulios consumida para procesos de activación del subarry.
    - *burstEnergy*, muestra en nanojulios la energía de burst que se ha consumido.
    - *writeEnergy*, muestra en nanojulios la energía que se ha empleado para escrituras.
    - *refreshEnergy*, muestra en nanojulios la energía que se ha empleado en refrescos.
    - *cancelledWrites*, número de escrituras canceladas.
    - *cancelledWriteTime*, tiempo empleado en escrituras canceladas.
    - *pausedWrites*, número de escrituras pausadas.

- *averagePausesPerRequest*, media de pausas por petición.
- *measuredPauses*, número de operaciones de pausa medidas.
- *averagePausedRequest*, media de las peticiones de pausa.
- *measuredProgresses*, número de peticiones de progresos.
- *reads*, número de lecturas realizadas en el subarray.
- *writes*, número de escrituras realizadas en el subarray.
- *activates*, número de activaciones en el subarray.
- *precharges*, número de precargas en el subarray.
- *refreshes*, número de refrescos en el subarray.
- *worstCaseEndurance*, peor factor de durabilidad en el subarray.
- *averageEndurance*, media de durabilidad de los elementos del subarray.
- *actWaits*, número de activaciones de espera.
- *actWaitTotal*, número total de activaciones de espera.
- *actWaitAverage*, media de activaciones de espera.
- *worstCaseWrite*, peor tiempo para realizar un write.
- *num00Writes*, número de escrituras de tipo 00.
- *num01Writes*, número de escrituras de tipo 01.
- *num10Writes*, número de escrituras de tipo 10.
- *num11Writes*, número de escrituras de tipo 11.
- *averageWriteTime*, tiempo medio de escrituras.
- *measuredWriteTimes*, tiempo de escrituras medido.
- *mlcTimingHisto*, muestra un histograma empleando el estilo de un diccionario python referente a las células multinivel (MLC).
- *cancelCountHisto*, muestra un histograma empleando el estilo de un diccionario python referente al número de cancelaciones.
- *wpPauseHisto*, muestra un histograma empleando el estilo de un diccionario python referente al número de pausas.
- *wpCancelHisto*, muestra un histograma empleando el estilo de un diccionario python referente al número de cancelaciones.

2. Al bank:

- *bankEnergy*, muestra en nanojulios la energía consumida por el bank.
- *activeEnergy*, muestra en nanojulios la energía de activación consumida.
- *burstEnergy*, muestra en nanojulios la energía de burst empleada.
- *refreshEnergy*, muestra en nanojulios la energía de refresco.

- *bankPower*, muestra en vatios la energía empleada en el bank.
- *activePower*, muestra en vatios la energía de activación del bank.
- *burstPower*, muestra en vatios la energía de burst del bank.
- *refreshPower*, muestra en vatios la energía de refresco del bank.
- *bandwidth*, muestra el ancho de banda consumido por el bank en megabytes por segundo.
- *dataCycles*, número de ciclos en los cuales se lleva a cabo transferencia de datos.
- *powerCycles*, número de ciclos en los cuales el bank tiene energía.
- *utilization*, utilización del bank.
- *reads*, número de lecturas en el bank.
- *writes*, número de escrituras en el bank.
- *activates*, número de activaciones en el bank.
- *precharges*, número de precargas en el bank.
- *refreshes*, número de refrescos en el bank.
- *activeCycles*, ciclos de activación en el bank.
- *standbyCycles*, ciclos en los cuales el bank se encuentra en *stand by* (en estado de reposo a la espera recibir algún comando).
- *fastExitActiveCycles*, número de ciclos de *stand by* (reposo) de precarga de salida rápida.
- *fastExitPrechargeCycles*, número de ciclos de *stand by* (reposo) de precarga de salida rápida.
- *slowExitPrechargeCycles*, número de ciclos de precarga de apagado de salida lenta.
- *actWaits*, número de activaciones en espera.
- *actWaitTotal*, número total de activaciones en espera.
- *actWaitAverage*, media de activaciones de espera.
- *averageEndurance*, media de durabilidad de los elementos en el bank.
- *worstCaseEndurance*, peor factor de durabilidad en el bank.

3. Al rank:

- *totalEnergy*, energía total consumida en el rank en nanojulios.
- *backgroundEnergy*, energía en nanojulios consumida en *background* (de fondo).
- *activateEnergy*, enenrgía en nanojulios consumida en activaciones.
- *burstEnergy*, energía en nanojulios consumida en operaciones de burst.
- *refreshEnergy*, energía empleada en nanojulios para refrescos.
- *totalPower*, energía total en vatios consumida en el rank.

- *backgroundPower*, energía en vatios consumida en *background* (de fondo).
- *activatePower*, energía en vatios consumida en activaciones.
- *burstPower*, energía consumida en vatios en el burst.
- *refreshPower*, energía consumida en vatios en refrescos.
- *reads*, número de lecturas realizadas en el rank.
- *writes*, número de escrituras realizadas en el rank.
- *activeCycles*, número de ciclos de activación en el rank.
- *standbyCycles*, ciclos en los cuales el rank se encuentra en *stand by* (en estado de reposo a la espera recibir algún comando).
- *fastExitActiveCycles*, número de ciclos en desconexión activa.
- *fastExitPrechargesCycles*, número de ciclos de salida rápida de precarga.
- *slowExitCycles*, número de ciclos de salida lenta de precarga.
- *actWaits*, número de activaciones de espera.
- *actWaitTotal*, número total de activaciones de espera.
- *actWaitAverage*, media de activaciones de espera.
- *rrdWaits*, número de esperas realizadas para asegurar la integridad de los datos referentes al parámetro de configuración *tRRDR*.
- *rrdWaittotal*, espera total realizada para asegurar la integridad de los datos referentes al parámetro de configuración *tRRDR*.
- *rrdWaitAverage*, media de la espera realizada para asegurar la integridad de los datos referentes al parámetro de configuración *tRRDR*.
- *fawWaits*, número de esperas realizadas para asegurar la integridad de los datos referentes al parámetro de configuración *tRAW*.
- *fawWaitTotal*, espera total realizada para asegurar la integridad de los datos referentes al parámetro de configuración *tRAW*.
- *fawWaitAverage*, media de la espera realizada para asegurar la integridad de los datos referentes al parámetro de configuración *tRAW*.

4. Al canal:

- *mem_reads*, lecturas de memoria.
- *mem_writes*, escrituras de memoria.
- *rb_hits*, peticiones a una misma fila del buffer.
- *rb_miss*, peticiones a otra fila del buffer.
- *averageLatency*, media de las latencias del canal.
- *averageQueueLatency*, media de las latencias de la cola.

- *averageTotalLatency*, media total de las latencias.
- *measuredLatencies*, latencias medidas.
- *measuredQueueLatencies*, latencia de las colas medidas.
- *measuredTotalLatencies*, latencia total medida.
- *simulation_ cycles*, ciclos de simulación realizados en el canal.
- *wakeupCount*, contador de operaciones de *wake up*.

5. Generales:

   - *TotalReadRequests*, total de peticiones de lectura.
   - *TotalWriteRequests*, total de peticiones de escritura.
   - *SuccessfulPrefetches*, operaciones exitosas de prefetcher.
   - *UnsuccessfulPrefetches*, operaciones fracasadas de prefetcher.
   - *Cycle*, número de ciclos totales de la simulación en frecuencia de CPU.

Para aquellos controladores que van ampliando la complejidad podemos encontrar un mayor número de parámetros:

1. FRFCFS, para este controlador encontramos además los siguientes parámetros referidos al canal:

   - *starvation_ precharges*, número de precargas realizadas correspondientes a recuperación de peticiones por inanición.
   - *write_ pauses*, peticiones de escritura pausadas para realizar una petición de lectura.

2. FRFCFS-WQF, para este controlador encontramos además los siguientes parámetros referidos al canal:

   - *rq_ rb_ hits*, peticiones en la cola de lectura a una misma fila del buffer.
   - *rq_ rb_ miss*, peticiones en la cola de lectura a otro fila del buffer.
   - *wq_ rb_ hits*, peticiones en la cola de escritura a una misma fila del buffer.
   - *wq_ rb_ miss*, peticiones en la cola de escritura a otro fila del buffer.
   - *total_ drains*, número total de drenajes.
   - *total_ drain_ writes*, número total escrituras en los drenajes.
   - *average_ writes_ per_ drain*, media de escrituras por drenaje.
   - *minimum_ drain_ writes*, mínimo de escrituras en un drenaje.
   - *maximum_ drain_ writes*, máximo de escrituras en un drenaje.
   - *total_ drain_ cycles*, número total de ciclos durante los que se ejecutaban drenajes.
   - *average_ drain_ cycles*, media de los ciclos de un drenaje.
   - *minimum_ drain_ cycles*, mínimo número de ciclos durante un drenaje.

- *maximum_ drain_ cycles*, máximo número de ciclos durante un drenaje.
- *total_ non_ drain_ cycles*, número total de ciclos durante los cuales no se ha ejecutado un drenaje.
- *average_ drain_ spacing*, media de ciclos entre drenajes.
- *minimum_ drain_ spacing*, mínimo número de ciclos entre drenajes.
- *maximum_ drain_ spacing*, máximo número de ciclos entre drenajes.
- *total_ read_ cycles*, número total de ciclos de lecturas.
- *average_ read_ spacing*, media de ciclos entre lecturas.
- *minimum_ read_ spacing*, mínimo número de ciclos entre lecturas.
- *maximum_ read_ spacing*, máximo número de ciclos entre lecturas.
- *total_ readqueue_ size*, total de peticiones de lectura al iniciar drenajes.
- *average_ predrain_ readqueue_ size*, media de peticiones en la cola de lectura antes del drenaje.
- *minimum_ predrain_ readqueue_ size*, mínimo de peticiones en la cola de lectura antes el drenaje.
- *maximum_ predrain_ readqueue_ size*, máximo de peticiones en la cola de lectura antes del drenaje.
- *total_ reads_ during_ drain*, número total de lecturas durante los drenajes.
- *average_ reads_ during_ drain*, media de lecturas durante los drenajes.
- *minimum_ reads_ during_ drain*, mínimas lecturas durante el drenaje.
- *maximum_ reads_ during_ drain*, máximas lecturas durante el drenaje.

7.5. Decoder

Como ya se ha indicado antes, el trabajo del decoder es realizado por dos objetos, el *TranslationMethod* y el *AddressTranslator*. Cuando se genera la estructura de árbol también se crea el decoder, por ello, desde el objeto *nvmain* en el método *SetConfig* se genera el objeto decoder, como el parámetro *Decoder* no está indicado en el archivo de configuración no se utiliza el *DecoderFactory*, sino que se emplea el *AddressTranslator*, además crea un objeto *TranslationMethod* con el cual establece el tamaño de los bits para direccionar empleando el método *SetBitWidths* como puede verse a continuación en la tabla 7.1.

Tras esto se emplea el método *SetCount* del *TranslationMethod* el cual almacena el parámetro junto con el tamaño total que tiene, esto se corresponde con las columnas 1 y 2 de la tabla 7.1.

El siguiente paso en la creación del decoder es generar el esquema de mapeo de direcciones, el cual se realiza con el método *SetAddressMappingScheme*, en este se asocian los diferentes componentes con un identificador numérico en el cuál los más significativos tienen un identificador mayor, un ejemplo de como se traduce la siguiente cadena “R:RK:BK:CH:C” con el esquema de mapeo de direcciones puede verse en la tabla 7.2 (no es necesario indicar el subarray en el

Parámetro	Tamaño	Bits
rowBits	8192	13
colBits	512	9
bankBits	8	3
rankBits	1	0
ChannelBits	4	2
SubarrayBits	1	0

Cuadro 7.1: Parámetros junto con su máximo tamaño y los bits necesarios para codificarlos.

archivo de configuración, lo añadirá después en el código). Este orden en el mapeo de direcciones se almacena para su futuro uso mediante el método *SetOrder*, pero con una particularidad, se almacena empezando por cero, no como puede verse en la tabla 7.2, por lo que todos los identificadores se decrementan en una unidad para ser almacenados.

Componente	identificador
Row	6
Rank	5
Bank	4
Channel	3
Column	2
Subarray	1

Cuadro 7.2: Ejemplo de esquema de mapeo de direcciones.

Tras esto el *TranslationMethod* tiene la configuración necesaria y puede ser asociado al objeto *AddressTranslator*, esta operación se realiza mediante *SetTranslationMethod* y se establece este objeto como el decoder utilizando *SetDecoder* (método contenido en *NVMObject*) para poder ser utilizado durante la simulación.

Podemos destacar los siguientes métodos de decoder:

- *Translate*. Se emplea cuando a partir de una petición (sobrecarga del método realizando la llamada nuevamente empleando la dirección física de la petición) o dirección física de una petición se quiere obtener el identificador relativo a la fila, la columna, el canal, el bank, el rank y el subarray. Este método calcula en primer lugar las potencias de dos referentes a los bits del offset del bus, los bits del burst y los bits de la columna más bajos (los cuales se calculan empleando la anchura del bus y el burst). Tras esto lleva a cabo un truncamiento de la dirección física de memoria empleando estos valores que acaba de calcular, después de suprimir esos bits entrará en un bucle para recorrer todos los elementos de la dirección en el cual:

  1. Empleando el método *FindOrder* encuentra el componente con los bits menos signifi-

cativos (en cada iteración se pasa al siguiente componente a la izquierda del anterior).

2. Con el método *Modulo* se extrae de la dirección de memoria física los bits correspondientes al componente identificado en la acción anterior.
3. Por último se realiza un desplazamiento a la derecha de los bits para eliminar los correspondientes al componente que ya se ha cargado y proseguir con los siguientes.

- *ReverseTranslate*. Se encarga de hacer el proceso contrario al anterior método, generando la dirección física a partir del valor de la fila, la columna, el canal, el rank, el bank y el subarray. Este método calcula en primer lugar las potencias de dos referentes a los bits del offset del bus, los bits del burst y los bits de la columna más bajos (los cuales se calculan empleando la anchura del bus y el burst), y empleando estos valores se crea una variable de desplazamiento para añadir a la dirección la anchura del bus y los bits de la columna más bajos. Tras esto se entrará en un bucle para recorrer todos los componentes que formarán la dirección (la columna, la fila, el rank, el bank, el canal y el subarray), en el cual:

  1. Empleando el método *FindOrder* encuentra el componente con los bits menos significativos (en cada iteración se pasa al siguiente componente a la izquierda del anterior).
  2. En función del componente se genera la parte de la dirección física que identifica a este componente en la dirección física completa. Para ello, en primer lugar se incrementa la dirección física de memoria añadiendo la dirección física identificativa de este componente y después se aumenta el desplazamiento para su utilización en la siguiente iteración como puede verse a continuación:

```
1 Dirección física += Componente * Desplazamiento
2 Desplazamiento <<= BitsComponente
```

  Cuando finaliza el bucle significa que se han añadido todos los componentes a la dirección física, que es lo que termina devolviendo este método.

Para no realizar este proceso en todo momento, tanto la dirección física como los identificadores referentes a la fila, la columna, el rank, el bank, el canal y el subarray se almacenan en un objeto de tipo *NVMAddress* para poder acceder a ellos de manera rápida.

Ejemplo de traducción de direcciones del decoder

A continuación se explica empleando un ejemplo el método *ReverseTranslate*, como ya se indicó el método *Translate* corresponde con el proceso contrario. Para ello se obtendrá la dirección de memoria física correspondiente a:

- Canal = 3.
- Rank = 0.
- Bank = 1.

- Columna 0.
- Fila = 1.
- Subarray 0.

En primer lugar se añaden al desplazamiento la anchura del bus, por el cual el desplazamiento es igual a 8, y tras esto se añaden los bits correspondientes a los bits menos significativos, con lo que este desplazamiento queda igual a 64. A partir de este punto se pasa a identificar el orden a la hora de incluir los componentes en la dirección física de memoria, por lo indicado en el archivo de configuración el orden será subarray, columna, canal, bank, rank y fila, tal y como se indica en la tabla 7.2. Además de incluir estos componentes en la dirección de memoria, se emplean los bits indicados en la tabla 7.1 para aumentar el desplazamiento:

- Subarray = 0, como el subarray es 0 tan solo se incrementa el desplazamiento, como el número de bits para poder diferenciar el subarray también es 0 este desplazamiento no se incrementa.
- Columna = 0, la dirección física tampoco se incrementa porque es la columna 0, pero en este caso se emplean 9 bits para redireccionar las columnas, por lo que el desplazamiento se incrementa a 32768.
- Canal = 3, se incrementa la dirección física con la fórmula descrita anteriormente (componente*desplazamiento, lo que significa 3 * 32768), por lo que la dirección física pasa a ser 98304. El desplazamiento se incrementa empleando los 2 bits de canal.
- Bank = 1, se incrementa la dirección física (1 * 98304) quedando esta en 131072. El desplazamiento también se incrementa empleando los 3 bits de bank.
- Rank = 0, como tanto el componente seleccionado es el 0 como los bits que se emplean para redireccionar el rank son 0, ni la dirección de memoria se incrementa ni el desplazamiento.
- Fila = 1, se incrementa la dirección física (1 * 1048576) quedando esta en 1277952. El desplazamiento también se incrementa con los 13 bits de fila, pero en este caso no es relevante ya que se termina la traducción.

De esta manera se ha pasado de los componentes canal 3, rank 0, bank 1, columna 0, fila 1 y subarray 0 a la dirección física 1277952 que los identifica.

7.6. Ejecución de ciclos desde el programa principal

En esta sección se describirá la ejecución del método *Cycle* correspondiente al objeto *GlobalEventQueue* junto con otros métodos que se ejecutan desde este.

Este método es llamado desde el bucle del programa principal y es el responsable de avanzar en los ciclos de la simulación para que se resuelvan las peticiones.

En este método se ejecutan el numero de ciclos que recibe por argumento, y en cada iteración obtiene el ciclo en el que sucede el siguiente evento (*GetNExtEvent*), si el siguiente evento sucede después de los ciclos, se avanza el número de ciclos recibidos como argumento y finaliza. En caso contrario lleva a cabo un avance de ciclos de grano fino (método *Loop*), en el cual va avanzando el ciclo actual y si en algún momento el ciclo actual coincide con el ciclo de ejecución de un evento de la lista de eventos llama al método *Process*. En las siguientes subsecciones se puede ver más en detalle la lógica de estos métodos.

7.6.1. *GlobalEventQueue::Cycle*

Se ejecutan de cero a n iteraciones:

- Se recupera el ciclo en el que ocurre el siguiente evento (*GetNExtEvent*).
- Se calcula la diferencia entre el ciclo del siguiente evento y el actual.
- Si la diferencia es mayor que las iteraciones restantes del bucle:
  1. Se avanza hasta el final.
  2. Se sincroniza.
  3. Se termina.
- Se calcula nuevamente la diferencia entre el ciclo del siguiente evento y el actual.
- Se ejecutan esos ciclos mediante grano fino (*Loop*).
- Se avanza la diferencia en las variables que almacenan el ciclo actual.
- Se sincroniza.

7.6.2. *GlobalEventQueue::GetNextEvent*

Se emplea un iterador para recorre un objeto de tipo *eventQueues*:

- Se calcula el multiplicador de frecuencia.
- Calcula el ciclo de evento global (ciclo de memoria).
- Se busca el evento que ocurre antes y se almacena en una variable que se devuelve.

7.6.3. *EventQueue::Loop*

Si el numero de iteraciones que recibe es cero significa que el procesamiento de eventos es en el ciclo actual, no se va a entrar al bucle *while*, por lo que:

- Se procesa (*Process*).
- Se termina.

En el caso que sea mayor que cero se entra al bucle *while* donde se recorren ese número de ciclos:

- Si en el total del número de ciclos no se van a procesar eventos se avanza esos ciclos y se termina.
- Si hay eventos a procesar se avanza hasta el ciclo de los eventos y se procesa (*Process*).

7.6.4. *EventQueue::Process*

En primer lugar se comprueba que el mapa de eventos de entrada no este vacío. Mediante un iterador se recorre el mapa de eventos desde el inicio hasta el fin:

- En función del tipo de evento:
  1. *EventCycle*, hace la llamada al método *Cycle* correspondiente al controlador de memoria.
  2. *EventIdle*, *EventRequest*, *EventUnknown* y *Default*, no hace nada.
  3. *EnventResponse*, hace la llamada al método *RequestComplete* correspondiente al controlador de memoria.
- Elimina el elemento al que apunta el iterador.

7.7. Caracterización del simulador

En esta sección se describirá el resultado de las diferentes ejecuciones del script A.2.1. La intención del empleo de este script no es llevar a cabo una evaluación del simulador NVMain, sino probar el funcionamiento del mismo y como se comporta al modificar los parámetros del archivo de configuración. Para esta tarea se han seleccionado los parámetros referentes a la política de gestión del cerrado de páginas (*ClosePage*) y el tipo de data rate (tasa de datos, *RATE*) empleado. Con respecto a este último, está estrechamente relacionado con el burst de datos, por lo que para proveer la misma cantidad de datos y poder comparar los resultados al modificar este parámetro es necesario modificar también el burst de datos:

- Si se emplea un data rate doble (caso por defecto), no alteramos el burst de datos ($tBURST = 4$).
- Si se emplea una data rate único, se debe duplicar el burts de datos ($tBURST = 8$).

Por lo tanto se ha lanzado el script seleccionando:

1. *ClosePage* con las opciones:
   - 0, la fila se cierra hasta que ocurra un fallo del buffer de la fila.
   - 1, la fila se cierra si no hay otra petición al buffer.
   - 2, la fila se cierra de manera inmediata.

2. *RATE* con las opciones:

    - 1, tasa de datos única.
    - 2, tasa de datos doble.

En la figura 7.12 se puede observar la ejecución del script y la información que muestra al usuario para saber en qué estado se encuentra en referencia al simulador y al procesamiento de los datos.

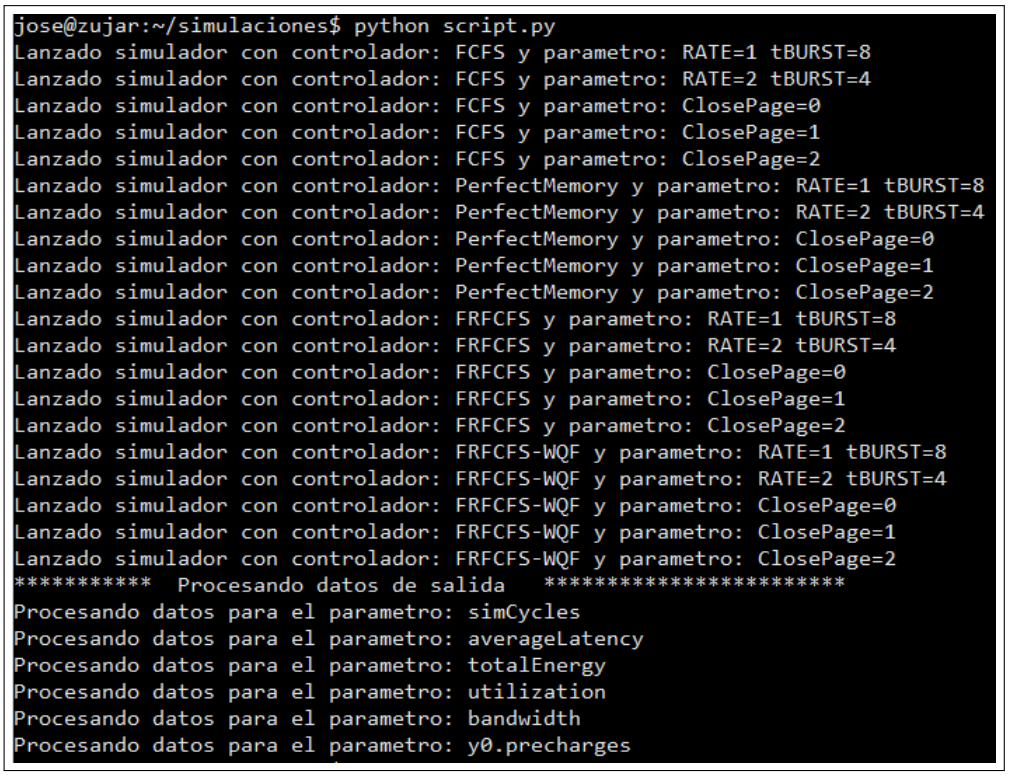

Figura 7.12: Ejecución del script

A continuación se muestran los resultados del parámetro *ClosePage* en las figuras 7.13, 7.14, 7.15, 7.16, 7.17 y 7.18. En estas imágenes encontramos una leyenda en el lateral derecho del gráfico, el significado de esta leyenda es la siguiente:

- *FFB*, cerrada hasta Fallo de la Fila del Buffer (0).
- *NAB*, No hay otro Acierto a la fila del Buffer (1).
- *CI*, Cerrado Inmediato de la fila del buffer (2).

Con respecto a los controladores de memoria, el controlador PerfectMemory tan solo muestra valores para los ciclos del simulador y la energía total, ya que este controlador se considera un nodo final son ningún tipo de retrasos y que finaliza las peticiones en el momento de llegada. Además se ha tomado el controlador FCFS como base para realizar la normalización con respecto a los otros controladores de memoria.

Aunque los valores obtenidos no pueden utilizarse para realizar ninguna evaluación como ya se indicó anteriormente, si que muestran unos resultados que tienen sentido, cuanto mayor es el número total de ciclos de simulación, se tiene un menor ancho de banda, mayor latencia y menor utilización.

Como se puede ver en la figura 7.13, la política *CI* muestra el peor caso en rendimientos, esto se debe a que al realizar el cerrado inmediato de kas filas del buffer es necesario volver a realizar la precarga de los datos para llevar a cabo las operaciones lo que se acaba traduciendo en un mayor número de ciclos de simulación.

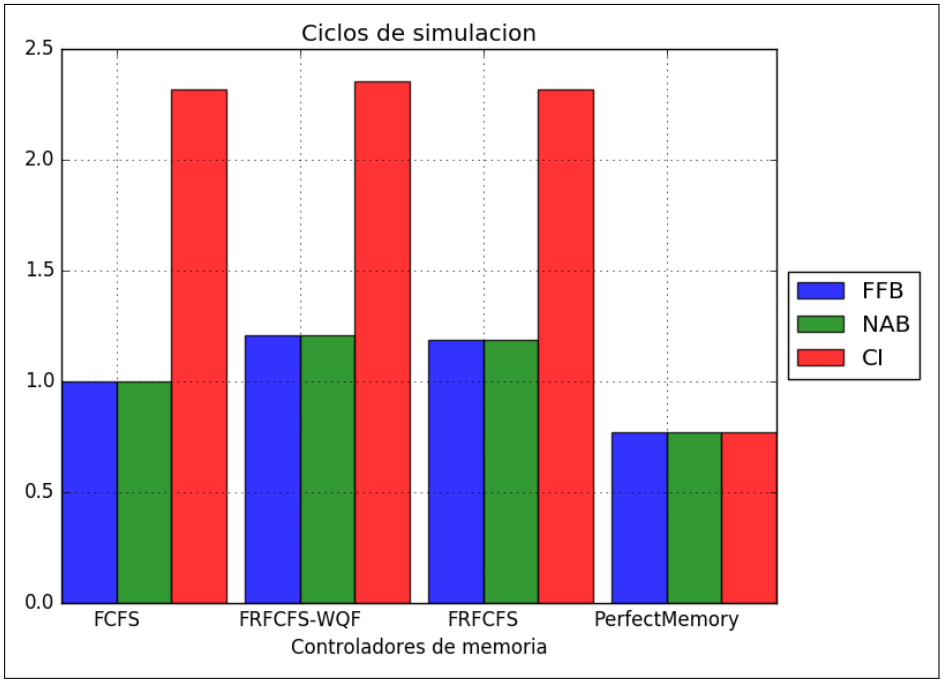

Figura 7.13: Ciclos de simulación

En la figura 7.14 puede observarse el ancho de banda obtenido para cada simulador. Aquellos casos que necesitaron un mayor número de ciclos para realizar la simulación en la figura 7.13 tienen un peor ancho de banda, esto tiene sentido, pues como el archivo de trazas es el mismo, los datos que se van a transmitir también son los mismo pero en un tiempo mayor.

En la figura 7.15 se muestra la energía que se consume por los componentes del simulador, aquellos casos en los cuales se tardó un mayor número de ciclos (figura 7.13) consumen mayor energía.

Con respecto a la latencia promedio obtenida (figura 7.16), las políticas que establecen un cierre del buffer de página más estricto generan una latencia mayor, pues se deben realizar un mayor número de precargas para acceder a los datos.

Como ya se ha indicado anteriormente y puede verse en la figura 7.17, el número de precargas se dispara con las políticas que tienen un mayor cerrado de páginas.

La utilización de banks (figura 7.18) es mayor cuanto mejores resultados se obtuvieron en el gráfico correspondiente a la figura 7.13.

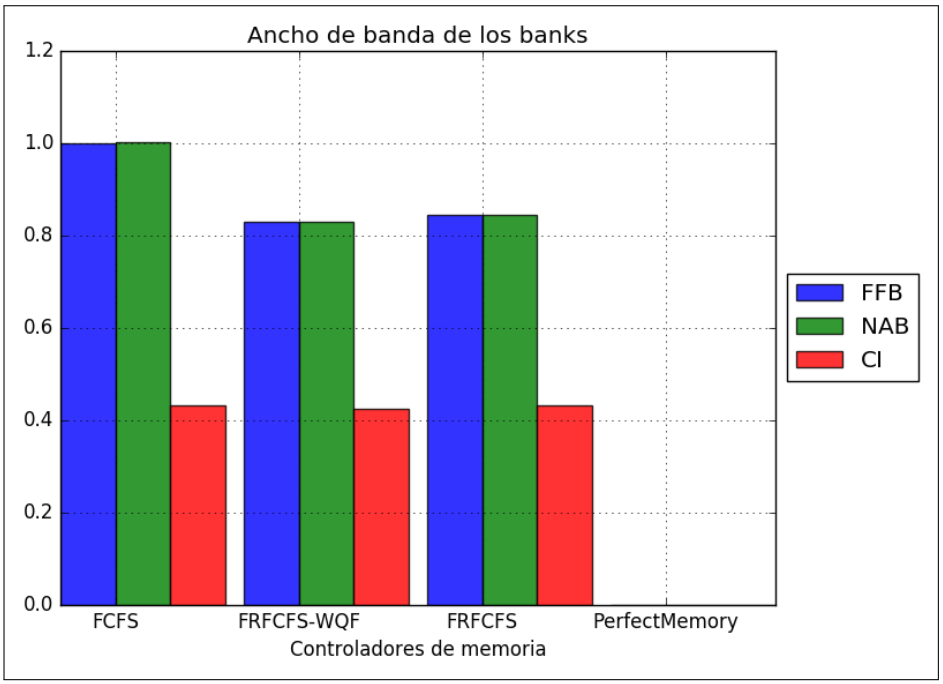

Figura 7.14: Ancho de banda

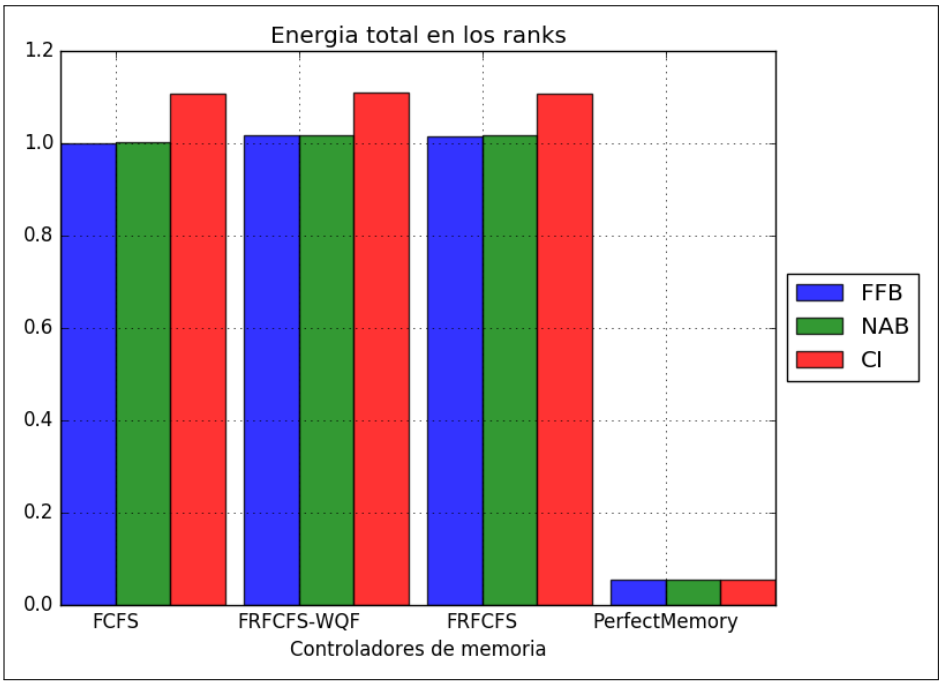

Figura 7.15: Energía total

En las figuras 7.19, 7.20, 7.21, 7.22, 7.23 y 7.24 se muestran los resultados correspondientes al parámetro del data rate. La leyenda situada a la derecha de los gráficos corresponde con:

- *DDR*, Doble Data Rate (tasa de datos doble, 2).
- *SDR*, Single Data Rate (tasa de datos única, 1).

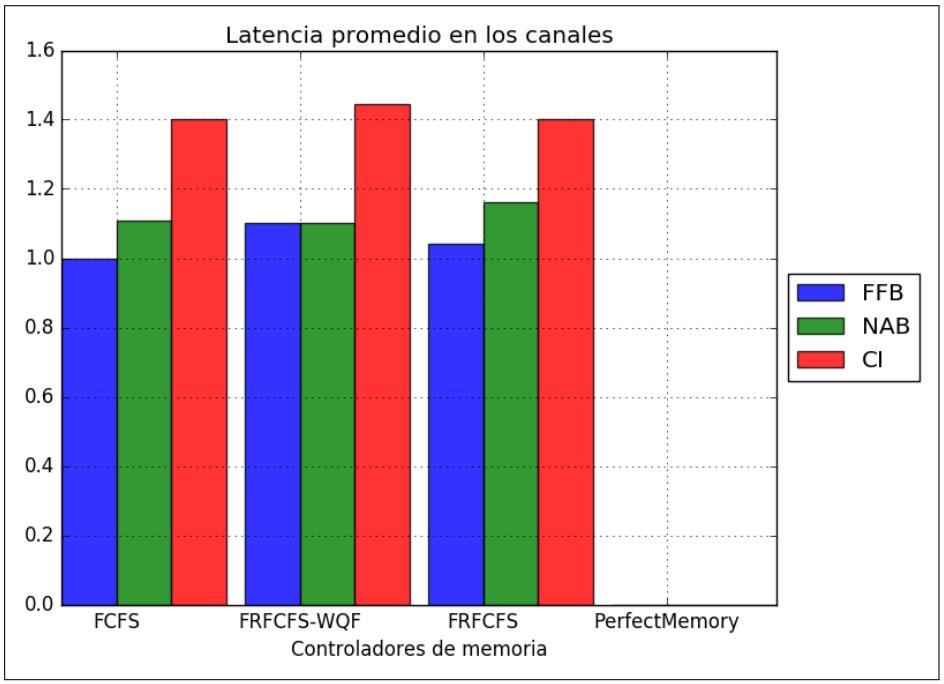

Figura 7.16: Latencia promedio

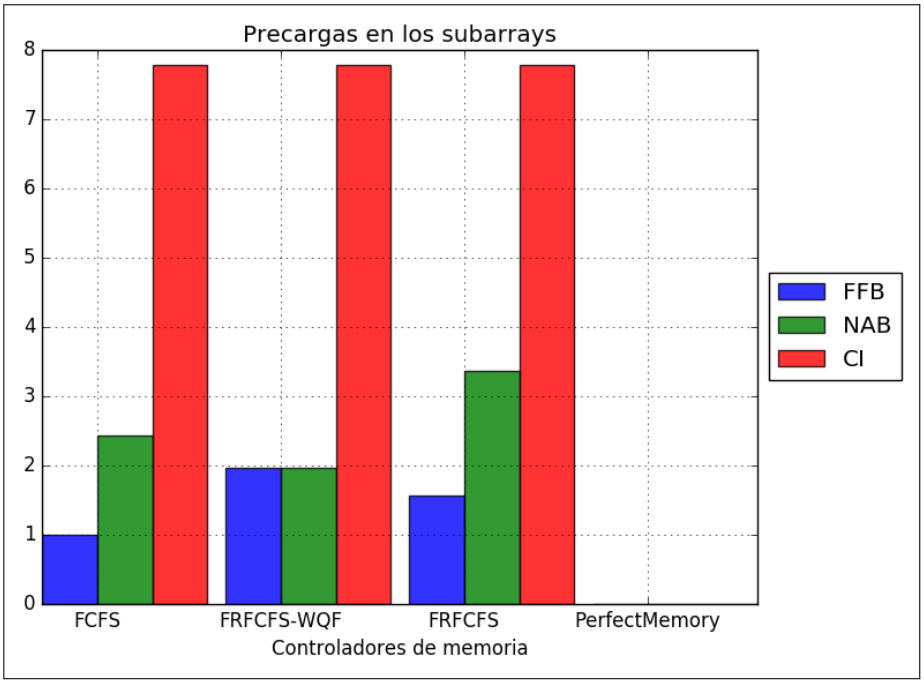

Figura 7.17: Precargas

La tendencia que se indicó en los gráficos anteriores se mantiene demostrando la fiabilidad del simulador.

Con respecto a esta política, como se puede ver en la figura 7.19 se muestra peor rendimiento cuando se emplea el SDR y se amplia el burst de datos, ya que son necesarios un mayor número de ciclos para poder transmitir los mismos datos.

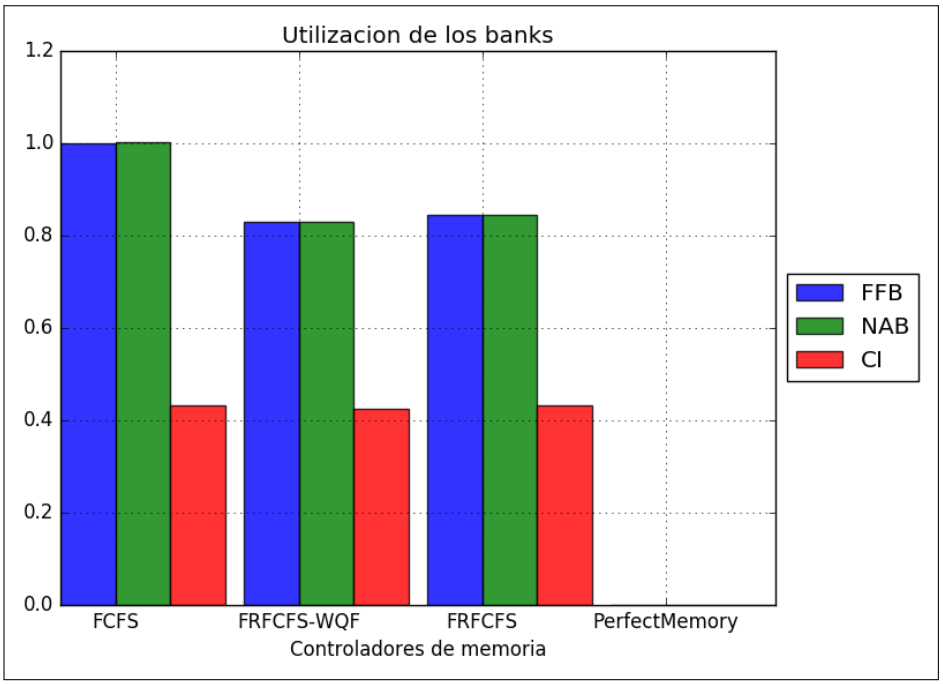

Figura 7.18: Utilización

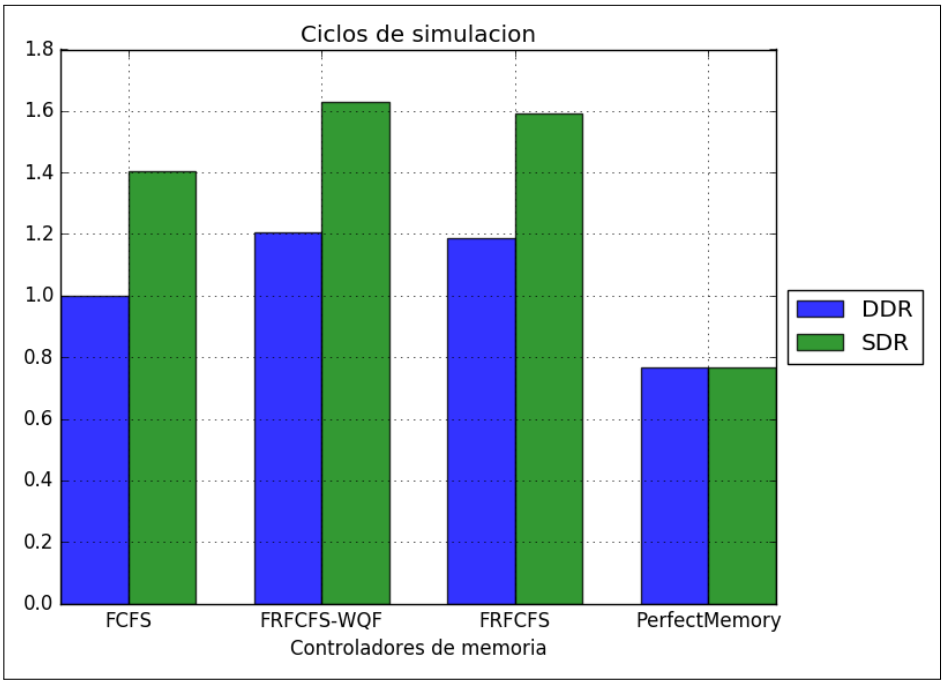

Figura 7.19: Ciclos de simulación

En la figura 7.20 puede observarse el ancho de banda obtenido para cada simulador. De la misma manera que antes, los casos que necesitaron un mayor número de ciclos para realizar la simulación en la figura 7.19 tienen un peor ancho de banda, ya que el archivo de trazas que se utiliza es el mismo y por lo tanto los datos a transmitir también son los mismos pero en un tiempo mayor.

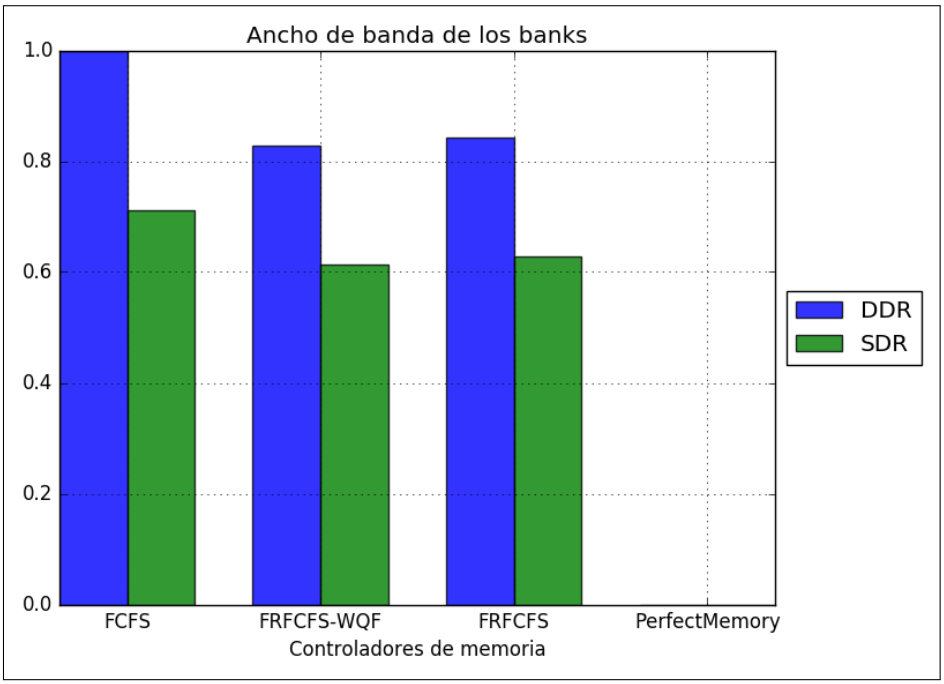

Figura 7.20: Ancho de banda

En la figura 7.21 se muestra la energía total consumida en el simulador, como el número de ciclos (figura 7.19) de las diferentes políticas no varia en exceso obtenemos unos valores muy similares de energía consumida, siendo un poco superiores aquellos casos que necesitaron un mayor número de ciclos.

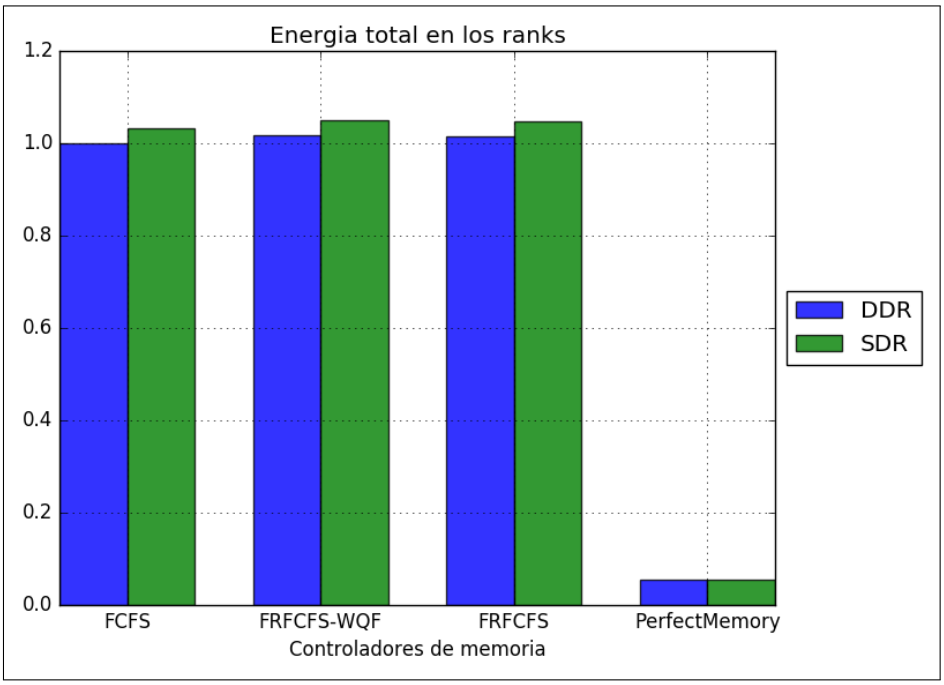

Figura 7.21: Energía total

La latencia promedio obtenida que puede verse en la figura 7.22 muestra como la política *SDR* tiene unos tiempos mayores, al tener que necesitar un número mayor de ciclos para resolver las peticiones se acumulan un número mayor de latencias.

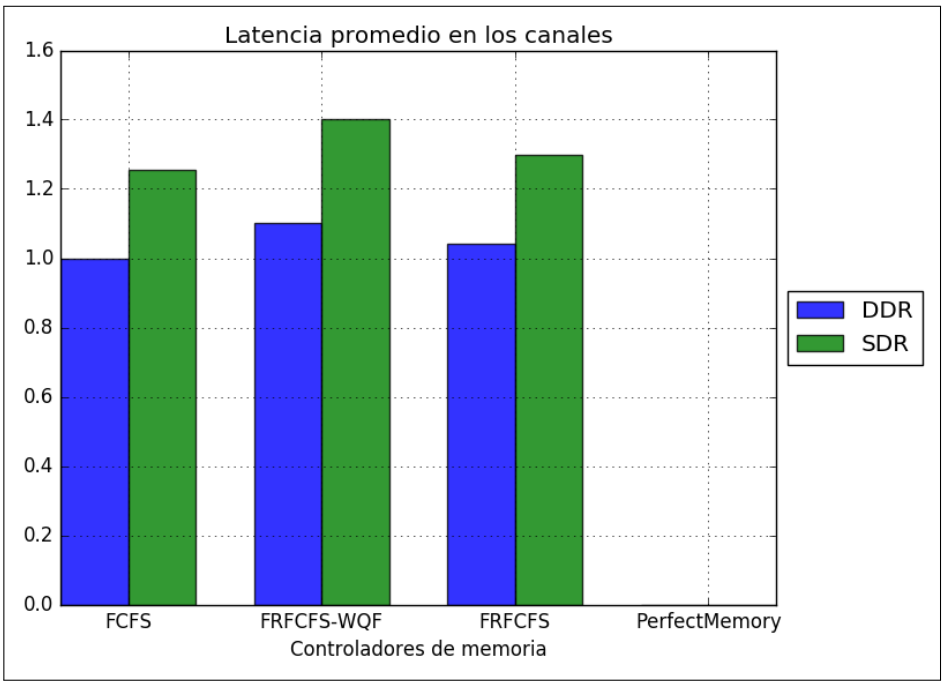

Figura 7.22: Latencia promedio

Con respecto a las precargas (figura 7.23), en este caso las precargas para los diferentes controladores de memoria son prácticamente las mismas aunque se cambie de política, esto se debe a lo ya mencionado anteriormente, se necesitan un mayor número de ciclos para enviar los datos, pero las precargas no se ven influidas por esto.

De manera distinta a la anterior política, se obtiene una mayor utilización de los banks (figura 7.24), ya que el número de operaciones a realizar no son las mismas, hay más transferencias de un menor número de datos.

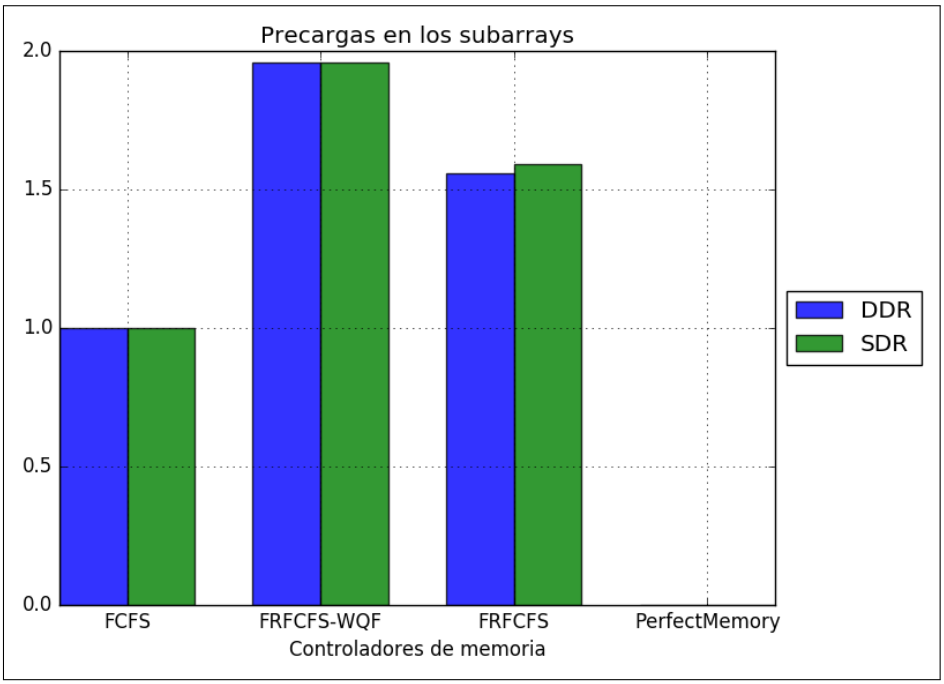

Figura 7.23: Precargas

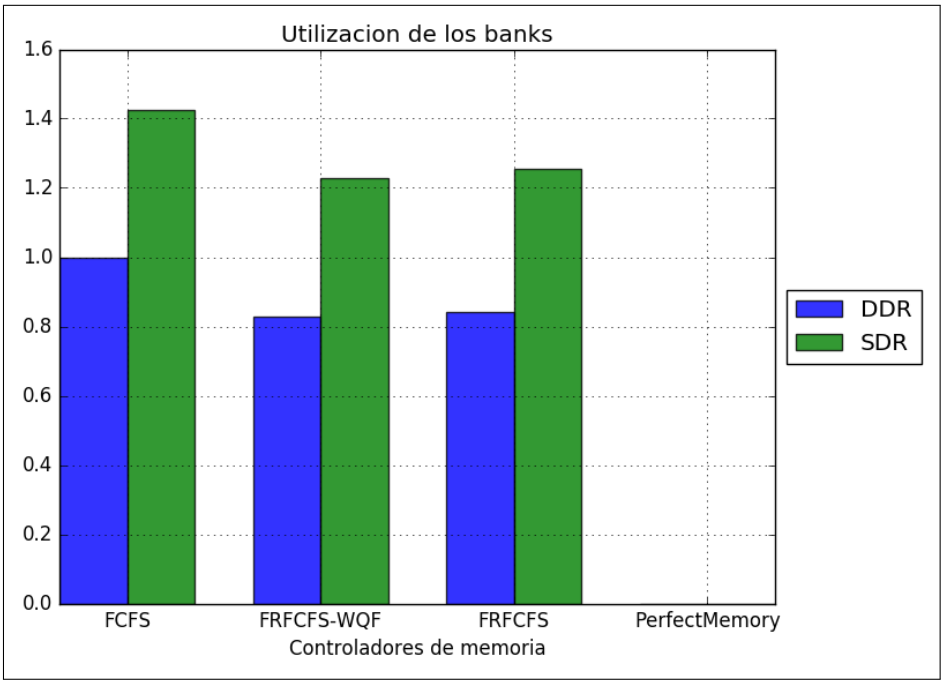

Figura 7.24: Utilización

7.8. Crear nuestros propios objetos

Aunque no se trata del procedimiento seguido en este trabajo final de máster, ya que ha consistido en realizar un estudio del simulador, es posible añadir nuestros propios objetos a NVMain. Como ejemplo, podemos añadir nuestro propio Rank para que sea utilizado en lugar

de *StandardRank*, que es el que emplea NVMain. Para ello, tan solo debemos crear un directorio en la ruta *nvmain/Ranks* con el nombre NuestroRank (nombre de ejemplo que se empleará) que contenga los archivos:

- *.cpp*
- *.h*

Los cuales implementaran los métodos necesarios por NVMain (métodos que pueden encontrarse en *StandardRank*, o métodos nuevos que se emplearan en otros objetos que creemos). Tras esto es necesario modificar *RankFactory* ya que es el objeto al que NVMain llama cuando necesita generar los objetos de tipo Rank en concreto el método *CreateRank* de la siguiente manera:

```
1  Rank *RankFactory::CreateRank( std::string rankName )
2  {
3      Rank *rank = NULL;
4
5      if( rankName == "StandardRank" ) rank = new StandardRank( );
6      else if( rankName == "NuestroRank" ) rank = new NuestroRank();
7      //else if( rankName == "CachedRank" ) rank = new CachedRank( );
8
9      return rank;
10 }
```

Y por último tan solo es necesario añadir en el archivo de configuración la línea:

```
1  RankType NuestroRank
```

Porque este parámetro no está definido en el archivo de configuraciópn y NVMain emplea por defecto *StandardRank*, al añadir esa línea se indica a NVMain el rank a utilizar distinto del rank por defecto.

Parte IV

Conclusiones

Capítulo 8

Valoración del trabajo realizado

8.1. Objetivos cumplidos

Este Trabajo de Fin de Máster tenía como principal objetivo llevar a cabo un estudio del simulador NVMain. El estudio realizado ha consistido en obtener un conocimiento superficial del funcionamiento global del simulador y un estudio más en profundidad sobre los controladores de memoria, permitiendo entender la simulación desde sus inicios y cuales son los diferentes pasos que se realizan desde que una petición llega al sistema hasta que se considera resuelta.

Con respecto a los objetivos marcados para este trabajo, podemos concretar que se han cumplido:

- Estudio de las diferentes memorias.
- Estudio general del simulador NVMain.
- Estudio de los diferentes controladores de memoria.
- Caracterización del simulador mediante ejecuciones empleando distintas políticas.
- Primera toma de contacto con la creación de objetos propios para su utilización en el simulador.

8.2. Competencias adquiridas

Durante el periodo de tiempo en el cual se ha realizado este trabajo, se han adquirido competencias relacionadas con el funcionamiento de las memorias, la programación en C++, ha sido posible trabajar en un grupo de investigación de la Universidad de Valladolid y se han realimentado los contenidos cursados durante este máster.

Se ha llevado a cabo la planificación de plazos para un desarrollo concreto, valorando los conocimientos previos para poder abordar tal tarea. Junto con las competencias adquiridas me han permitido mejorar mis habilidades y poder encarar de mejor manera mi futuro profesional.

El estudio del simulador se ha basado completamente en emplear la ingeniería inversa para poder comprender las tareas que se realizaban durante las simulaciones, durante la carrera y el

máster no se encuentra una asignatura como tal que se base en estos principios, si que encontramos numerosas asignaturas de programación que han servido para poder entender la lógica del simulador. Para la planificación y la gestión del proyecto ha sido importante lo impartido en la signatura de planificación y gestión de proyectos.

8.3. Conclusiones

Las conclusiones de este proyecto son las siguientes:

1. El simulador NVMain esta desarrollado empleando programación orientada a objetos y un sistema de módulos muy escalable que facilita su modificación o la inclusión de nuevos objetos para sustituir a los existentes.

2. La fase correspondiente al estudio del simulador es la que tiene mayor importancia, ya que se trata del principal objetivo del proyecto, y la que mayor tiempo ha requerido.

3. Como durante la realización de este trabajo el grupo de trabajo solo ha consistido en una persona, cualquier aparición de un riesgo provoca que aumente la duración del trabajo.

4. Los riegos de un proyecto son determinantes para la finalización del mismo. Pueden traducirse en un gran sobrecoste o en el fracaso de este.

5. Llevar a cabo el trabajo dentro de un grupo de investigación hace que sea similar a llevar a cabo una tarea en un grupo de trabajo real, siendo necesario la obtención de resultados en los periodos de tiempo establecidos.

6. La realización de un documento académico como este permite aprender a desarrollar un documento técnico para futuros trabajos.

8.4. Trabajo futuro

Como trabajo futuro queda la parte que no ha podido ser realizada durante el desarrollo de este trabajo, la modificación o creación de un nuevo controlador de memoria que introduzca alguna característica relevante para el estudio, como podría ser el control de las escrituras sobre los diferentes subarrays y la distribución de esta carga para conseguir una mayor durabilidad de las memorias.

También es posible una evaluación del simulador más allá de la caracterización que se ha realizado, para poder llevar a cabo una comparación de los controladores de memoria con respecto a los rendimientos obtenidos, ya que el archivo de trazas empleado no es suficientemente significativo como para que los resultados sean comparables.

Parte V

Apéndices y Bibliografía

Apéndice A

Contenido del CD-ROM y manuales

A.1. Contenido del CD-ROM

En el CD-ROM adjunto se incluye la memoria en formato PDF y el script Python para lanzar las simulaciones.

A.2. Manual de usuario

A.2.1. Script para generar las gráficas

Para lanzar de manera automática las simulaciones, recoger sus datos y generar las gráficas correspondientes se emplea el script [16] [15] que se describe a continuación, pero para poder ejecutarlo los directorios descritos en las variables *TMPDIR* y *OUTDIR* deben existir, así como las rutas que referencian:

- Al ejecutable.
- Al directorio de archivos de trazas.
- Al archivo de configuración

Para indicar los parámetros a estudiar se debe modificar la lista *COMPARA* añadiendo o eliminando los elementos que contiene respetando la estructura indicada en el script. Si lo que se desea es cambiar las políticas de estudio, esto se realiza en la función *anyadirParam*, donde se añaden a una lista las políticas que se sobreescribirán con respecto al archivo de configuración. En caso de querer añadir o quitar algún controlador de memoria de las simulaciones a realizar se debe modificar la variable *controladores*. El comportamiento del script consiste en dos fases principales:

1. Fase de simulación, se lanzan las simulaciones correspondientes a las diferentes políticas para los controladores de memoria indicados y almacena la salida de estas simulaciones en diferentes subdirectorios dentro del directorio *TMPDIR*.

2. Fase de procesamiento, partiendo de los archivos generados anteriormente se obtienen los datos para los parámetros que se van a estudiar diferenciando por las distintas políticas. Se normalizan los valores y se generan las gráficas, las cuales se almacenan en los subdirectorios correspondientes a la política a la que corresponden dentro del directorio *OUTDIR*.

```
#!/usr/bin/python
# -*- coding: latin-1 -*-

# Paquetes a importar
import os
import re

from subprocess import PIPE, Popen

import matplotlib

# Se emplea Agg ya que no se va a utilizar la salida estandar
matplotlib.use('Agg')

import matplotlib.pyplot as plt
import numpy as np

# Ruta del ejecutable
EXEC = "/home/jose/nvmain/nvmain.fast "

#Ruta donde se encuentran los archivos de trazas
DIRNVT = "/home/jose/nvmain/Tests/Traces/"

# Ruta donde se encuentra el archivo de configuracion
CONF = "/home/jose/nvmain/Config/RRAM_ISSCC_2012_4GB.config "

# Ciclos a simular, 0 es simulacion completa
CYCLE = " 0 "

# Directorio de salida temporal
TMPDIR = "/home/jose/simulaciones/tmp/"

# Directorio de salida final
OUTDIR = "/home/jose/simulaciones/salida/"

# Pie de imagen para los graficos
LABELX = 'Controladores de memoria'

# Parametros a recoger
# Estructura [Parametro, Comando de seleccion bash, Texto para el titulo]
COMPARA = [ ['simCycles', '" | cut -d " " -f 4', 'Ciclos de simulacion'], ['
    averageLatency', '" | cut -d " " -f 2', 'Latencia promedio en los canales'], ['
    totalEnergy', '" | cut -d " " -f 2 | cut -d "n" -f 1', 'Energia total en los
    ranks'], ['utilization', '" | cut -d " " -f 2', 'Utilizacion de los banks'], ['
    bandwidth', '" | cut -d " " -f 2 | cut -d "M" -f 1', 'Ancho de banda de los
    banks'], ['y0.precharges', '" | cut -d " " -f 2', 'Precargas en los subarrays']
     ]


# Colores para las diferentes politicas del grafico
COLORS = ['b','g','r','y']

```

```
47  # Clase para almacenar los parametros que se remplazaran del archivo de
        configuracion
48  class parametro():
49    name = ""
50    value = []
51    valueName = []
52
53    def __init__(self, nam, vals, names):
54      self.name = nam
55      self.values = vals
56      self.valueName = names
57
58  # Ejecuta un comando bash y captura la salida de dicho comando
59  def terminal(comando):
60      process = Popen(args=comando, stdout=PIPE, shell=True )
61      return process.communicate()[0]
62
63  # Comprueba que el archivo de entrada tenga extention .nvt
64  def checkNvt(archivo):
65    match = re.search(r'.nvt$', archivo)
66
67    if match and not os.path.isdir(archivo):
68      return True
69    else:
70      return False
71
72  # Parametros que se sustituiran en el archivo de configuracion
73  # Politica por defecto la primera
74  def anyadirParam():
75    lst = []
76
77    param = parametro("RATE", ["2 tBURST=4", "1 tBURST=8"], ["DDR", "SDR"])
78    lst.append(param)
79
80    param = parametro("ClosePage", ["0", "1","2"], ["FFB", "NAB", "CI"])
81    lst.append(param)
82
83    return lst
84
85  # Comprueba que el directorio de entrada y sus subdirectorios existen, si no es asi
        los crea
86  def checkDir(param, directorio, bandera):
87
88    # Se comprueba si existe el directorio para los diferentes valores del parametro
89    if os.path.isdir(directorio+param.name):
90
91      # Se recorren los diferentes valores
92      for value in param.values:
93
94        # Se comprueba si existe el directorio para el valor concreto
95        # Controlando si tiene otro parametro asociado
96        valor = value.split(" ")[0]
97        if os.path.isdir(directorio+param.name+"/"+valor):
98
99          # Se eliminan todos los archivos del directorio
100         os.system("rm " + directorio+param.name+"/"+valor+"/*");
101
102
```

```
103         # Existe el directorio de salida
104         elif( bandera == 1 ):
105
106           # Se eliminan los archivos
107           os.system("rm " + directorio+param.name+"/*")
108
109           # Se sale, se evita ejecutar el comando sobre el mismo directorio otra vez
110           break
111
112         # No existe, se crea
113         else:
114           # Se opera sobre el directorio temporal
115           if(bandera == 0):
116             os.system("mkdir " + directorio+param.name+"/"+valor);
117
118
119     # No existe, lo crea
120     else:
121       os.system("mkdir " + directorio+param.name);
122       checkDir(param, directorio, bandera)
123
124
125
126 # Comprueba que los directorios existen y elimina los archivos antiguos para la
      nueva iteraccion
127 def checkDirs(lstParam):
128
129     # Se recorre la lista de parametros
130     for param in lstParam:
131
132       # Se hace la comprobacion de los directorios temporal y de salida
133       checkDir(param, TMPDIR, 0)
134       checkDir(param, OUTDIR, 1)
135
136 # Convierte la cadena de entrada en una lista y suma sus elementos si son varios
137 def suma(cadena):
138
139     # Genera una lista
140     lista = cadena.rstrip("\n").split("\n")
141
142     # inicializa la variable
143     resultado = 0
144
145     # Si hay mas de 1 elemento se obtiene su sumatorio
146     if(len(lista) > 1):
147       for item in lista:
148         resultado = resultado + float(item)
149
150     # Si solo hay un elemento se convierte a entero
151     elif len(lista) == 1:
152
153       # Si no es un caracter vacio
154       if lista[0] != '':
155         resultado = float(lista[0])
156
157     # Se devuelve el resultado
158     return resultado
159
```

```
160 # Se normalizan los elementos de las listas y se genera la lista objetivo
161 def norma(unidades, num):
162 
163   listaFinal = []
164   listaDivisores = []
165 
166   # Guardamos el divisor de la normalizacion para no perderlo en la primera
        iteraccion del siguiente bucle
167   divisor = unidades[0][0]
168 
169   # Se normalizan los valores de la lista en funcion de FCFS (primer lista)
170   for i in range(len(unidades)):
171     for j in range(len(unidades[i])):
172 
173       unidades[i][j] = unidades[i][j] / divisor
174 
175 
176   # Se crean las sublistas necesarias
177   for i in range(num):
178     listaFinal.append([])
179 
180   # Se pueblan las listas referentes a las politicas
181   for lista in unidades:
182     cont = 0
183     for item in lista:
184       listaFinal[cont].append(item)
185 
186       cont = cont + 1
187 
188   # Se devuelve la lista objetivo
189   return listaFinal
190 
191 
192 # Genera el grafico con los elementos de entrada
193 def grafBar(unidadesX, unidadesY, labelX, title, rutaSalida, valueName):
194 
195   # Numero de controladores de memoria
196   numGrupos = len(unidadesX)
197 
198   # Genera las listas con los valores en el orden necesario para el grafico
199   lstUnidades = norma(unidadesY, len(valueName))
200 
201   # Configuracion del grafico
202   fig, ax = plt.subplots()
203   index = np.arange(numGrupos)
204   opacity = 0.8
205 
206   # En funcion del numero de politicas se emplea diferente anchura
207   if(len(valueName) == 2):
208     bar_width = 0.35
209   elif(len(valueName) == 3):
210     bar_width = 0.3
211 
212 
213   # Se recorren las diferentes politicas
214   for indice in range(len(valueName)):
215 
216     # Grafico de barras
```

```
217       plt.bar(index + indice*bar_width, lstUnidades[indice], bar_width, alpha=opacity
              , color=COLORS[indice], label=valueName[indice])
218
219
220     # Configuracion del grafico
221     plt.xlabel(labelX)
222     plt.title(title)
223     plt.xticks(index + bar_width, tuple(unidadesX))
224     plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
225     plt.margins(x=0)
226     plt.grid()
227
228     # Se guarda y se cierra el grafico
229     plt.savefig(rutaSalida, bbox_inches='tight')
230     plt.close()
231
232
233   # Comprueba si la lista de entrada esta compuesta enteramente por ceros
234   def checkNula(lista):
235     bandera = True
236
237     for item in lista:
238       if float(item) != 0.0:
239         bandera = False
240
241     return bandera
242
243   # Programa principal
244   if __name__ == "__main__":
245
246     # Controladores de memoria a simular
247     controladores = parametro("MEM_CTL", ["FCFS", "PerfectMemory","FRFCFS","FRFCFS-
          WQF"], [])
248
249     # Lista de parametros a remplazar
250     lstParam = anyadirParam()
251
252     # Se comprueba si existen los directorios de salida temporales y finales
253     # Si no existen los crea
254     checkDirs(lstParam)
255
256     #Para cada controlador
257     for contrl in controladores.values:
258
259       # Lista los elementos del directorio NVT
260       files  = os.listdir(DIRNVT)
261
262       # Para cada archivo de traza
263       for nvt in files:
264
265         # En caso de ser un archivo de traza
266         esNvt = checkNvt(DIRNVT+nvt)
267         if esNvt:
268
269           # Se recorren los parametros a utilizar
270           for param in lstParam:
271
272
```

```
273             for value in param.values:
274               # Se controla que el valor no tenga otro parametro asociado
275               valor = value.split(" ")[0]
276
277               # Se genera el nombre del archivo
278               outFile = contrl+".txt"
279
280               # Se lanza la simulacion almacenando la salida
281               print "Lanzado simulador con controlador: " + contrl + " y parametro: "
                      + param.name + "=" + value
282               # en el directorio temporal asociado al parametro
283               os.system(EXEC + CONF + DIRNVT+nvt + CYCLE + controladores.name+'='+
                      contrl + " "+param.name+"="+value+" > "+ TMPDIR + param.name + "/"
                      + valor + "/" + outFile);
284
285
286
287   print "***********  Procesando datos de salida   ***********************"
288
289   # Para cada elemento a comparar
290   for item in COMPARA:
291
292
293     print "Procesando datos para el parametro: " + item[0]
294
295     # Se almacena el texto asociado a la escala X
296     labelX = LABELX
297
298     # Nombre del fichero de salida resultante
299     nomFich = item[2]
300
301     # Para cada politica de ejecucion
302     for param in lstParam:
303
304       # Se inicializan las variables para la iteracion
305       # Elemento que contendra una lista de listas con los valores de las
              diferentes politicas
306       unidades = []
307
308
309       # Para cada valor de esa politica
310       for value in param.values:
311
312         # Se inicializa la variable para la iteracion
313         # Variable para acceder a la lista correspondiente de la politica
314         index=0
315
316         # Se listan los archivos contendios en el directorio
317         # Se controla si tiene otro parametro asociado
318         valor = value.split(" ")[0]
319         files = os.listdir(TMPDIR + param.name + "/" + valor)
320         files.sort()
321
322         # Se alamacenan los indices del eje X
323         contro=[i.split('.')[0] for i in files]
324
325         # Se recorren los archivos para obtener los datos
326         for txt in files:
```

```
327
328             # Solo se anyade una lista la primera vez que se recorren los archivos
329             # Se evitan tener listas vacias
330             if value == param.values[0]:
331               # Se anyade un indice a la lista de listas
332               unidades.append([])
333
334             # Ruta al archivo
335             rutaEntrada = TMPDIR+param.name+"/"+valor+"/"+txt
336
337             # Se obtiene y almacena el valor asociado (si son multiples filas se
                    suman)
338             unidad =  suma(terminal("cat " + rutaEntrada + ' | grep "' + item[0] +
                    item[1] ))
339             unidades[index].append(unidad)
340
341             # Se incrementa la variable indice
342             index = index + 1
343
344
345         # String con la ruta de salida
346         rutaSalida = OUTDIR+param.name+'/' + nomFich + '-' +param.name+"="+value +'.
                png'
347
348
349         # Se genera el grafico y se almacena
350         grafBar(contro, unidades, labelX, nomFich, rutaSalida, param.valueName)
```

A.3. Diagramas

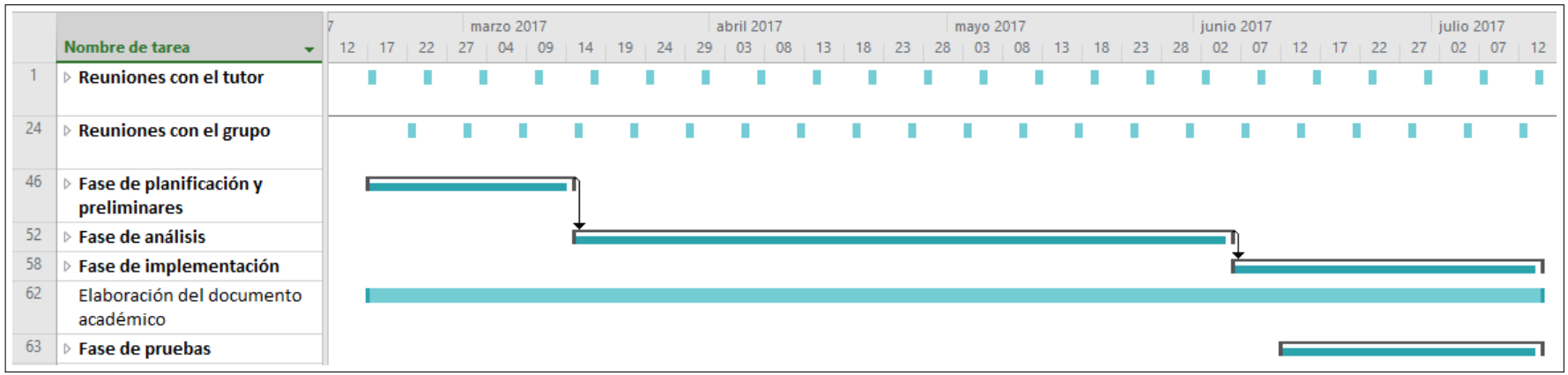

Figura A.1: Diagrama de Gantt de la planificación real del proyecto

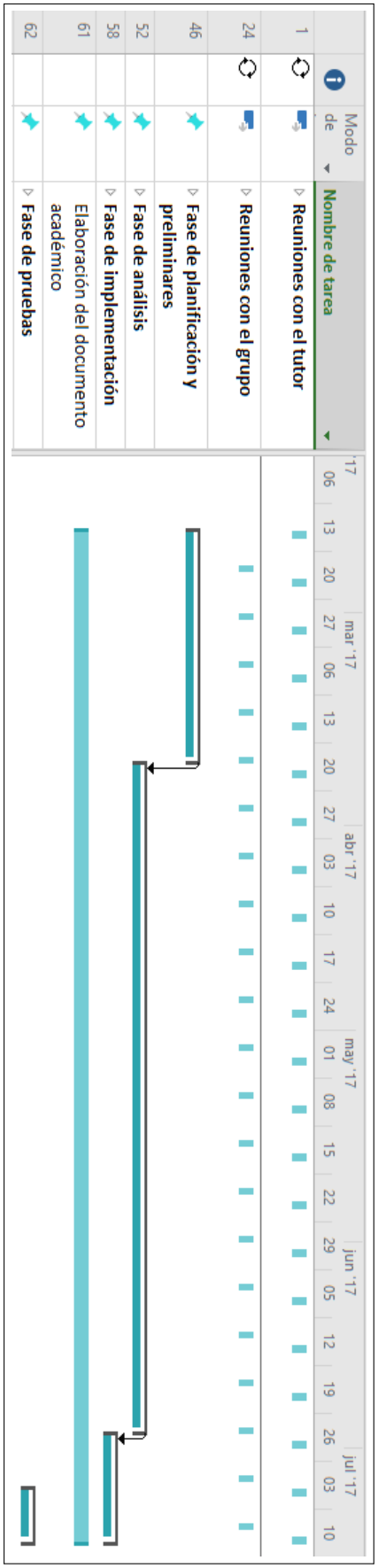

Figura A.2: Diagrama de Gantt de la planificación real del proyecto

Bibliografía

[1] Architectural modeling for emerging memory technologies. http://mrp5060.bgghost.net/ISCA14-Tutorial-NVMain-Merged-Version-0614.pptx. Fecha de último acceso: 2017-06-10.

[2] Burst mode access and timing. http://www.pcguide.com/ref/ram/timingBurst-c.html. Fecha de último acceso: 2017-04-23.

[3] Cacti. http://www.hpl.hp.com/research/cacti/. Fecha de último acceso: 2017-03-28.

[4] Conductancia y conductividad. http://e-ducativa.catedu.es/44700165/aula/archivos/repositorio/2750/2952/html/241_conductancia_y_conductividad.html. Fecha de último acceso: 2017-03-15.

[5] Corelink controllers and peripherals. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100000_0000_00_en/nde1393328371158.html. Fecha de último acceso: 2017-03-06.

[6] Distancia de hamming. https://es.wikipedia.org/wiki/Distancia_de_Hamming. Fecha de último acceso: 2017-05-10.

[7] Distribución normal. https://es.wikipedia.org/wiki/Distribuci%C3%B3n_normal. Fecha de último acceso: 2017-05-10.

[8] Funcionamiento del caching. http://www.ordenadores-y-portatiles.com/caching.html. Fecha de último acceso: 2017-03-25.

[9] The gem5 simulator. http://gem5.org/Main_Page. Fecha de último acceso: 2017-03-23.

[10] Memory timings explained w/ suggested timings memset vs. bios. https://forums.tweaktown.com/gigabyte/27283-memory-timings-explained-suggested-timings-memset-vs-bios.html. Fecha de último acceso: 2017-04-15.

[11] Memristor. https://es.wikipedia.org/wiki/Memristor. Fecha de último acceso: 2017-03-10.

[12] Mercurial. https://www.mercurial-scm.org/. Fecha de último acceso: 2017-03-23.

[13] Nvmain wiki. http://wiki.nvmain.org/. Fecha de último acceso: 2017-06-28.

[14] Nvsim main page. http://nvsim.org/wiki/index.php?title=Main_Page. Fecha de último acceso: 2017-03-28.

[15] Plotting commands summary. https://matplotlib.org/api/pyplot_summary.html. Fecha de último acceso: 2017-07-10.

[16] Python 2.7.13 documentation. https://docs.python.org/2.7/. Fecha de último acceso: 2017-07-10.

[17] Resistive random-access memory. https://en.wikipedia.org/wiki/Resistive_random-access_memory. Fecha de último acceso: 2017-03-20.

[18] Sdram tutorial. http://taututorial.yolasite.com/. Fecha de último acceso: 2017-03-07.

[19] Sense amplifier. https://en.wikipedia.org/wiki/Sense_amplifier. Fecha de último acceso: 2017-03-21.

[20] ¿aislante o metal? http://www.investigacionyciencia.es/revistas/investigacion-y-ciencia/la-autntica-revolucin-sexual-522/aislante-o-metal-8714. Fecha de último acceso: 2017-03-20.

[21] Siddheswar Maikap Amit Prakash, Debanjan Jana. Taox-based resistive switching memories: Prospective and challenges. Technical report, 2013.

[22] Kees Goossens Benny Akesson. *Memory Controllers for Real-Time Embedded Systems*. Springer-Verlag New York, NJ, USA, 2012.

[23] Spencer W. Ng David T. Wang, Bruce Jacob. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc. San Francisco, San Francisco, CA, USA, 2007.

[24] Kin Leong Pey. Design for reliability in resistive ram for ict-enabled devices. https://es.slideshare.net/dawnchia718/design-for-reliability-in-resistive-ram-for-ictenabled-devices. Fecha de último acceso: 2017-03-17.

[25] M. Poremba and Y. 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, Aug 2012.

[26] M. Poremba, T. Zhang, and Y. Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.

[27] Deepak C. Sekar. Resistive ram: Technology and market opportunities. http://microlab.berkeley.edu/text/seminars/slides/DeepakSekar.pdf. Fecha de último acceso: 2017-03-20.

[28] Rainer Waser, editor. *Nanoelectronics and Information Technology*. Wiley-VCH, New Jersey, USA, 2012.

[29] Yuan Xie, editor. *Emerging Memory Technologies*. Springer-Verlag New York, NJ, USA, 2014.