Propuesta 2:
SW Cache Coherence modelling and evaluation

David De la Hoz Aguirre
Ingeniería en Computadores
Cartago, Costa Rica
divad0907@estudiantec.cr

Daniela Brenes Otárola
Ingeniería en Computadores
Cartago, Costa Rica
dani08@estudiantec.cr

Kenichi Hayakawa Bolaños
Ingeniería en Computadores
Cartago, Costa Rica
kenichih48@estudiantec.cr

Oscar Méndez Granados
Ingeniería en Computadores
Cartago, Costa Rica
oscar.mendez@estudiantec.cr

***Index Terms*—PE, Cache, Bus, Memoria, MESI, MOESI, Coherencia, Snoop**

I. Propuesta de Diseño

Con base en el hecho de que las Caches son fundamentales para mejorar el desempeño de un processing element (PE), la implicación es que estas deben estar equipados para siempre proveer información correcta y actualizada. De aquí surge la necesidad de la coherencia de caché, el cual permite el manejo correcto de memoria compartida, y así garantizar el funcionamiento correcto de las operaciones realizadas por los diferentes PEs. El objetivo del proyecto es modelar y evaluar diferentes protocolos de coherencia de memoria, específicamente el MESI, MOESI y su impacto en el desempeño de las PEs.

La arquitectura implementada es la siguiente:

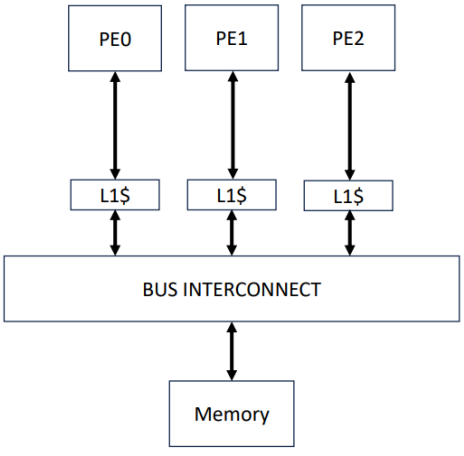

Fig. 1. Diagrama de la arquitectura implementada

Como se puede observar, el sistema es uno multiprocessor (MP), que consiste de 3 PEs, de los cuales a cada uno les corresponde un cache privado. Esas caches se comunican con un bus compartido, y una memoria principal única en la que se basan todas las caches. Cada componente del sistema es un elemento de hardware (HW) en principio, y al igual que en la vida real, esas funcionan concurrentemente y de forma independiente. Como todos son elementos independientes, y la idea es que cada PE se modela con un thread.

Los PEs envían escrituras, lecturas y hacen operaciones de incremento. El interconnect hace el enrutamiento y notificación de invalidación según el protocolo de coherencia que esté activo.

Con respecto a protocolos de coherencia, se implementaron 2, el MESI y el MOESI. Estos son acrónimos representativos de estados posibles que pueden tener datos dentro de una cache, y son los que permiten que caches que no se comunican directamente entre ellos mantengan información correcta para fines de uso.

- M: Modified, implica que la caché tiene la versión más actualizada de un dato, y esta aún no está en memoria principal
- E: Exclusive, implica que la caché es la única que actualmente tiene el dato cargado, y está igual de actualizado que con memoria principal
- S: Shared, implica que la caché no es la única que actualmente tiene el dato cargado, sino que está en este y en otras cachés también. Específicamente en el caso de MOESI, estos adicionalmente tienen la propiedad de ser read-only.
- I: Invalid, implica que la caché no tiene una versión válida del dato y es necesario ir a traer el dato de memoria, o de una caché que si esté cargado en una, en el caso específico del MOESI, en caso de estar cargado, el dato necesariamente debe venir del caché que tenga el dato en estado Owned, y no Shared.
- O: Owned, es un caso especial de Shared en donde este caché es el único que tiene permiso de modificar el dato, o comunicárselo a alguna otra caché o memoria principal. El estado Owned implica que puede o no existir otros caches con el mismo dato en Shared; sin embargo, no es conclusivo con respecto a esto. Este estado es

exclusivo al protocolo MOESI y es el que permite mayor flexibilidad con respecto a datos ”dirty” sin tener que obligatoriamente acceder a memoria principal cada vez.

La ventaja general que tiene el protocolo MOESI sobre el MESI es que el MOESI minimiza en lo posible los accesos a memoria principal, dado que esto es computacionalmente costoso. Sin embargo, viene al costo de que en términos de implementación, el MESI es superior, porque sus 4 estados son generales y no hay que preocuparse por la misma cantidad de casos específicos que hay en el MOESI.

Siempre que puedes usar el cache, es bueno, y el estado Owned del MOESI procura utilizar esto a su favor en la mayor parte posible, escribiendo a memoria principal únicamente en el último momento posible, esto siendo cuando el dato se va a sobreescribir por otro dato que requiere del mismo espacio en la cache. En rendimiento, esto es teóricamente una ventaja. Existen edge cases específicos en donde no sea el caso, pero todos se pueden amortiguar con agregar más líneas a la caché.

II. Especificación de componentes del sistema

A nivel más detallado, el sistema posee 3 PEs, los cuales pueden ejecutar 3 instrucciones: write addr reg, read addr reg, e incr reg. Estos corren en paralelo mediante el uso de threads, y por medio de la interfaz gráfica se puede hacer stepping o start para procesar instrucciones por cada PE. Este también permite generar código aleatorio para cada procesador y permite la selección del procolo que deseas implementar. Luego de haber finalizado la simulación, el sistema presenta un resumen de total de transacciones y su tipo, ya sean reads, writes o invalids, y se puede visualizar los contenidos de los registros de cada PE, la cache y la memoria.

El sistema en general se puede dividir en 4 partes: Conexión de PE con su Caché respectiva, Cache-Bus-Memoria, Logger e interfaz.

La primera parte es conexión de PE con su Caché. En esta etapa se realiza la comunicación entre estos dos elementos mediante requests de read y write que emite el PE. Este espera una respuesta de la caché la cual realiza procesos para poder comunicarse con otras partes del sistema. Al hacer lo necesario la caché retorna una respuesta al PE para que este pueda ejecutar la instrucción que se realizó. Esta parte del sistema es sencillo pero esencial.

Para el diseño de la caché, se decidió utilizar una implementación Direct-Mapped. Esto tiene sus ventajas y desventajas. Por un lado, la política de reemplazo es muy sencilla ya que cada línea de memoria solo puede estar *mapped* a una línea específica en la caché, lo que simplifica bastante la lógica de la implementación. Por otro lado, las cachés Direct-Mapped tienen el gran problema del *trashing*, debido a esta misma condición en las que las líneas de memoria tienen solo una posición en la cual colocarse en la caché, y debido a la localidad de memoria, en sistemas reales hay una gran posibilidad de frecuentemente reemplazar líneas con otras, lo que puede llegar a ser ineficiente. Sin embargo, al ser este proyecto una simulación y abstracción de un sistema real, no se incurre en programas complejos que causen *trashing* a nivel mayor, por lo que en este caso se decidió esta implementación.

Cuando el caché recibe una request del PE aquí es donde empieza el trabajo de la relación Cache-Bus-Memoria. En esta etapa del sistema es donde se realiza todo lo relacionado con lectura de memoria, write back policy, y lo más importante el controlador del protocolo de coherencia de cache. Debido a que el bus tiene acceso a todas las caches y además a memoria, se le dio la tarea de manejar los request hechos por la cache y manejar los snoops del protocolo seleccionado. La manera que se implementaron los estados del protocolo es mediante una máquina de estados, la cual el bus usa para poder saber que tipo de acción realizar: compartir datos entre cache, invalidar datos de otros caches, ir a memoria (leer y escribir).

La etapa del Logger se encarga de ir tomando mediciones de requests y el rendimiento del sistema en general. Este se activa cuando el bus o el cache realiza un request o un response, el cual lo guarda en un archivo para analizarlo después. Y finalmente la interfaz que es el medio que se usa para poder visualizar el comportamiento del sistema completo, viendo las caches y la memoria y cómo varía a cómo se ejecutan las instrucciones.

Este sistema completo se puede ver en la figura (2), donde se muestra las interacciones que tiene cada bloque con cada uno para poder entender el flujo que se lleva.

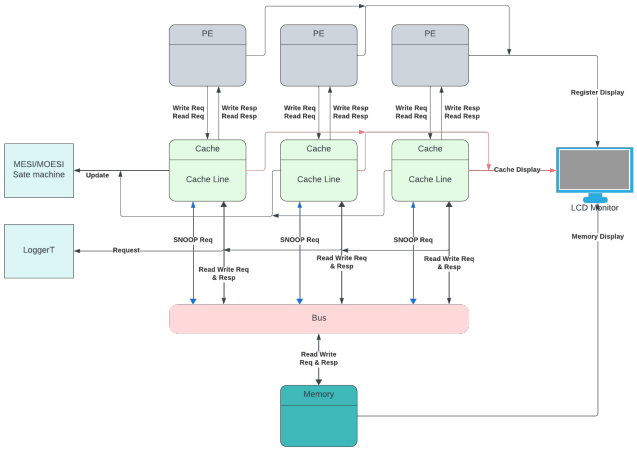

Fig. 2. Diagrama de contexto del sistema

III. Explicación del proceso de diseño

El grupo inicialmente modeló los diferente componentes como sus propias clases para tener bien modularizado los diferentes requerimientos y además para recrear el diagrama de arquitectura a nivel de código. Esto inició programando la memoria principal, el BusInterconnect, el Processing Element y dos clases para la cache: Cache y CacheLine. La idea de Cache es que fuera una lista de CacheLines, y así separar la

lógica interna de CacheLines de las funcionalidades propias de la Cache. Una vez implementado esto, el grupo debatió diferentes formas de conectarlo, partiendo de los módulos creados, cómo manejar el flujo y comunicación de datos. Esto fue el primer reto grande el proyecto. Se experimentaron con diferentes ideas, como integrando altamente las clases Cache y ProcessingElement, y de ahí comunicándolos con una clase teórica CacheController, el cual sería clase padre de dos clases MESI y MOESI, y que este administrara la comunicación entre el BusInterconnect y los Caches de los processing elements. Luego de unos intentos de implementar esta idea, la descartamos porque la alta dependencia entre las caches y los PEs estaba causando mucho problema y requería una implementación compleja del resto del sistema.

La solución era más bien independizar las clases individuales más, modularizándolos de una mejor manera, y a partir de esto, programar un archivo top que funcionara como el canvas general de nuestra arquitectura. En el top, instanciamos la cantidad de cada clase que necesitemos, para nuestra implementación específica instanciamos por ejemplo 3 PEs, pero otra ventaja de este approach es que la solución se generaliza de que si por x o y razón se pidiera un 4to PE, sería un asunto tan simple como instanciar otra PE. Esto aplica para cada componente diferente, y al estar todos en una misma clase top, este se encarga de comunicarlos de manera apropiada, y esto es una manera de abstraer nuestra arquitectura en un ratio casi de 1:1. Una vez sorteadas las comunicaciones, el siguiente tema era el manejo de los protocolos de coherencia de cache. Este se decició implementar mediante un StateMachine, en el cual se programaban todos los estados y sus transiciones, y en el BusInterconnect, se van comunicando las acciones de cada data a las cachés correspondientes para que estas transiciones de estado con base en el evento ocurrido, ya sea read/write hit/miss. Estas luego le comunican su respuesta al Bus para que este le comunique el ”snoop” o ”probe” a las demás cachés en donde estas también utilizan el mismo StateMachine para hacer sus cambios correspondientes. El protocolo es una variable en donde la lógica para cada uno está programada y revisa nada más a la hora de entrada, cuál es el protocolo actual.

Una vez implementado la funcionalidad core, Se necesitaba algo que estuviera accesible en cualquier parte del proceso, entonces se utilizó un singleton para que el logger fuera accesible de cualquier parte del código, y partiendo de este se hacían llamadas de log con base en los eventos como write request, read request, invalidation, memory write y memory read. Así es como el equipo puede obtener estadísticas de funcionamiento y las gráficas resumen al final con información relacionada al desempeño del diseño. La GUI no tiene lógica como tal, es un medio de comunicación entre el usuario y la lógica del programa como tal, en cuanto a lo gráfico, se hizo así partiendo de un hecho de que sea lo más intuitivo para el usuario y agradable a la vista.

A. *MESI*

Para la implementación del protocolo MESI, este tiene consideraciones mucho más generales respecto al MOESI, por lo que fue implementado primero. Su diagrama de estados es el siguiente:

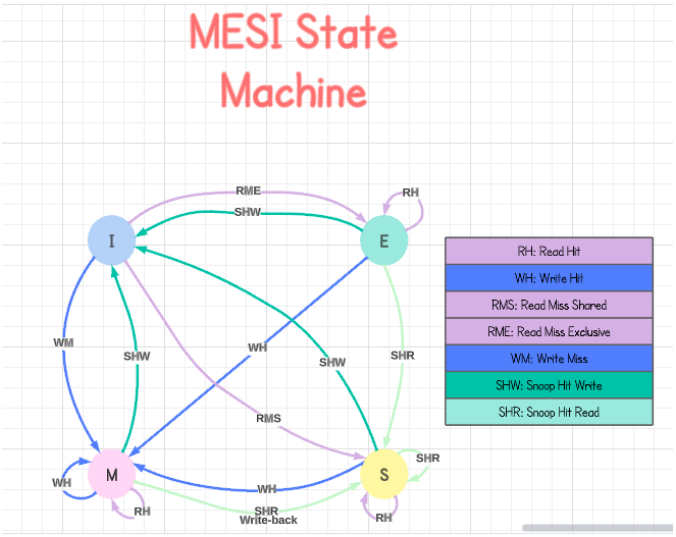

Fig. 3. Diagrama de estados del protocolo MESI

Las consideraciones importantes que hay que tener con el MESI es que el estado Exclusive implica que eres el único cache que tiene el dato actual cargado, y adicionalmente este es el mismo valor que está en memoria. Cuando escribes el dato para que tenga un valor nuevo, transiciones al estado modified, el cual significa que el dato actual aún no ha sido escrito a memoria, y que su caché contiene la versión más actualizada.

Si alguna otra caché lee el dato, su estado debe ser shared porque implica que el dato existe en al menos una otra cache, y si por ejemplo en el otro caché el dato es modified, este debe escribirse a memoria principal, y transicionar a shared también. Si se escribe en algun caché en donde el dato está shared, este debe pasar a modified, y mandar la señal al bus hacia las demás cachés que invaliden sus copias porque ya no son correctos. Si un dato es inválido, el cache debe buscarlo de otra caché en donde el dato ya esté cargado, y en caso de que no, debe ir a leer el dato a memoria principal.

B. *MOESI*

El protocolo de coherencia de cache MOESI es parecido en principio al MESI, es la máquina de estados del MESI con un estado más: Owned.

A continuación se puede observar el diagrama de estados del protocolo MOESI:

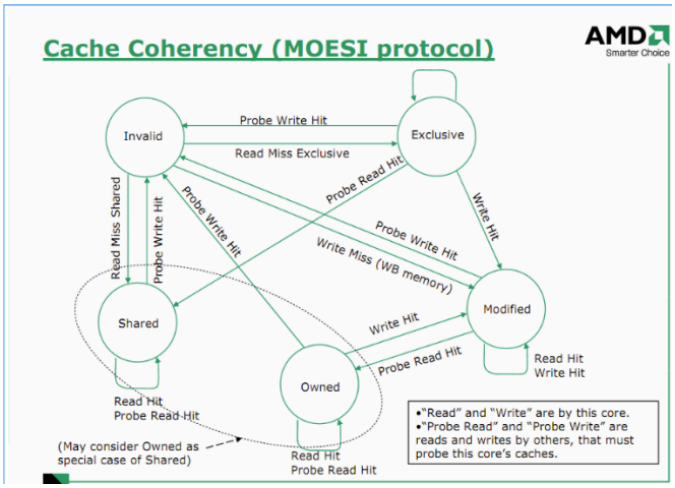

Fig. 4. Diagrama de estados del protocolo MOESI

Cabe notar que aunque los estados Modified, Exclusive, Shared e Invalid comparten el mismo nombre que en el protocolo anterior, hay implicaciones por debajo que son importantes tener en mente a la hora de hacer una implementación, y estas sutilezas tienen implicaciones bastante significativas a nivel de comportamiento y a nivel de código.

Lo importante que hay que tener en cuenta siempre es que el MOESI busca minimizar accesos a memoria principal, dado que si un mismo dato está en caché, y en memoria, acceder el dato desde una caché será más rápido por un margen significativo. Es decir, en la medida de lo posible, el protocolo MOESI busca que si un dato está en una caché, entonces que venga desde una caché y no de memoria. De ahí surge el estado Owned, es el estado en donde un bloque de caché tiene el permiso exclusivo de escribir el dato en otros lados, y de modificarlo.

El Owned se puede considerar como un caso especial del shared. Cuando un dato está en el estado Owned, puede haber o no copias del dato en otras cachés, esto no se puede garantizar ni negar partiendo únicamente del estado Owned en una caché, pero inversamente si en alguna caché se encuentra algún dato en estado shared, necesariamente existe en alguna otra caché el mismo dato con el estado Owned. Cualquier estado Shared en el MOESI es read-only, y si alguna cache decide leer el dato, y no está en él, el que debe proveer el dato es la caché que lo tiene en el estado Owned.

El estado Owned es como un Shared con privilegios extra, y este estado se le brinda a la primera caché que cargó el dato.

La existencia de este estado Owned permite tolerar data que es ”dirty” sin tener la obligación de siempre que hay un desfase entre una caché y memoria, tener que ir a escribir el dato a memoria. De hecho, la única instancia que se escribe en memoria en el MOESI, es cuando el bloque que contiene algún dato en el estado Modified u Owned va a ser sobreescrito por otra dirección al que le corresponde el mismo bloque en el direct-mapped caché, por lo que ahí sí se ve obligado a escribir en memoria. De lo anterior, se puede concluir que en ninguna transición entre estados se escribe a memoria, lo cual garantiza parte de una mejora en desempeño en el protocolo.

El estado Exclusive es el más similar a su contraparte en el MESI, el estar en exclusive implica que tanto la memoria como el dato actual en caché son iguales, y además únicamente se encuentra cargado en el caché actual. Si este dato se modifica mediante una escritura, pasa al estado modified, si es leido por otra caché, más bien pasa al estado Owned, y la caché que hizo la lectura lo lee en estado Shared.

El estado Modified también es parecido a como es en el MESI, con la gran diferencia de que si alguna otra caché lee el mismo dato, este no escribe a memoria y transiciona a Shared, si no que pasa al estado Owned sin escribir en memoria, y le provee el dato a la caché que lo está intentando de leer. De acá se observa directamente un ahorro de tiempo respecto al MESI.

De la misma manera, el estado Invalid implica que el caché tiene que buscar el dato, ya sea de una caché que tenga el dato cargado o en caso de que no lo esté, desde memoria principal.

El estado Shared únicamente puede ser leído, y en caso de que desee escribir un dato que está en el estado Shared, formalmente lo que ocurre es que el caché que tiene el dato en shared le comunica al caché que lo tiene en Owned a través del bus para que se le conceda el estado de Owned, y el que anteriormente estaba en Owned, pasa a ser Shared. Una vez que en el caché actual se obtiene el estado de Owned, este se puede sobreescribir y pasa al estado Modified, y manda la señal para invalidar el dato en todas las demás cachés que lo tenga en Shared. De aquí, en nuestra implementación, se implementó una abstracción que la escritura pasa de Shared a Modified, y se invalidan todas las demás copias del dato, ya sea en Shared u Owned. El resultado final de la realidad y esta abstracción es la misma, y a nivel de implementación se ahorra complejidades en el código.

IV. Evaluación de desempeño de los 2 protocolos

Después de simular varios ejemplos de códigos pudimos obtener varios reportes de los resultados de las simulaciones utilizando ambos protocolos y se observaron varias cosas. Lo claro que se pudo ver entre ambos métodos fue que el nivel de complejidad de MESI es mucho menor que el de MOESI. Esto se debe a el estado extra que trae MOESI, pero además a que tiene muchas más condiciones y casos específicos a considerar a la hora de implementarlos.

Ya revisando la como tal métricas de rendimiento podemos ver una clara diferencia en la cantidad de memory writes que se hacen entre los 2 protocolos. Analizando varios resultados se pudo observar como la cantidad de memory writes disminuye al usar MOESI compactado con el MESI por una cantidad significante (*Depende del código). En los ejemplos que siguen (**??**fig:MESTTest)) (6), que son solo una de las varias pruebas realizadas, donde se puede ver como la cantidad de MEM writes en MESI es de 5 mientras que en MOESI es de 1; en términos de rendimiento esto es algo que se debe tomar en cuenta. Este comportamiento en realidad es algo esperado debido a la naturaleza de MOESI ya que este solo hace writeback a memoria cuando este es reemplazado en la cache y el estado es Ownes o Modified, mientras que MESI hace write back siempre que se pase de Modified a shared y también cuando se reemplaza en caché.

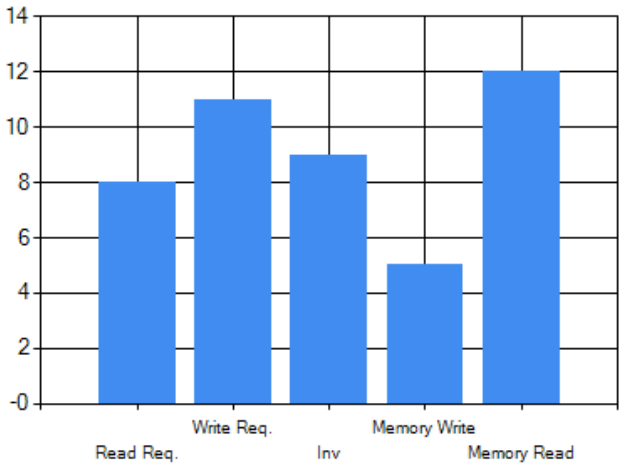

Fig. 5. Reporte de simulación MESI

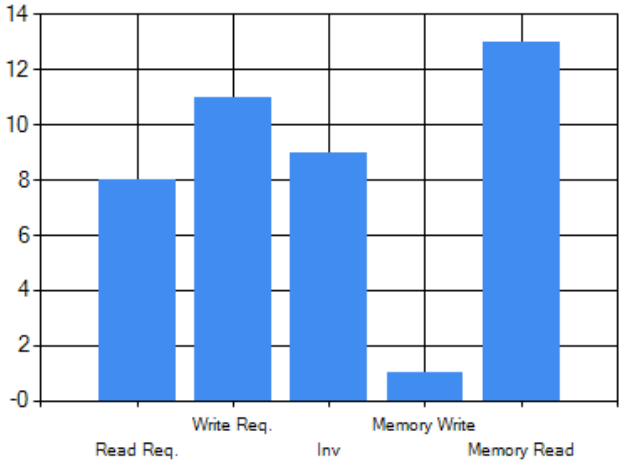

Fig. 6. Reporte de simulación MOESI

En el caso de las otras métricas no cambian mucho debido a que en ambos protocolos se deben realizar reads y writes a cache constantes solo que cambia cuando se escribe a memoria. Una métrica que sí podría y en efecto cambia es la de invalidaciones. Esto es debido a que al tener otro estado de Owned, el MOESI invalida más otros espacios de caché que comparten ese addres en comparación con MESI.

REFERENCES

[1] A. Ibarra, YouTube, 2022.MOESI Simulator. https://www.youtube.com/watch?v=RM5kBHsW9Eo

[2] H. Weatherspoon, "Caches (writing) - department of computer science," Caches (Writing), https://www.cs.cornell.edu/courses/cs3410/2013sp/lecture/18-caches3-w.pdf.

[3] "Cache," cache, https://people.cs.pitt.edu/ xianeizhang/notes/cache.html.

[4] S. Dey, "Design and implementation of a simple cache simulator in Java" Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols, https://www.researchgate.net/profile/Somdip-Dey/publication/263004594_Design_and_Implementation_of_a_Simple_Cache_Simulator_in_Java_to_Investigate_MESI_and_MOESI_Coherency_Protocols/links/5405ebd70cf23d9765a79c4b/Design-and-Implementation-of-a-Simple-Cache-Simulator-in-Jav pdf

[5] B. H. Juurlink, YouTube, 2018. 4 2 4 MESI and MOESI Protocols. https://www.youtube.com/watch?v=nrzT044qNIc

[6] "CS 61C summer 2017– Cache Coherency - University of California, Berkeley," CS 61C Summer 2017– Cache Coherency, https://inst.eecs.berkeley.edu/ cs61c/su18/disc/11/Disc11Sol.pdf.