Protocolos de coherencia en sistemas multiprocesador

José Daniel Montoya Salazar
jd.montoya.s8@gmail.com
Instituto Tecnológico de Costa Rica
Área académica de Ingeniería en Computadores
Arquitectura de Computadores II

***Resumen*—This document describes the investigation process, implementation and results of a simulated multiprocessor system about cache coherence protocols related to states.**

The system was written using Python and simulates a MOESI cache coherence protocol, on a system with four processors executing instructions simultaneously. In addition, the system has a single memory that must be shared by the four processors and is accessed through a bus, the latter is monitored by a snoopy within each processor. As writing policie, the implemented system uses write-through so in this way the cache and memory are always updated.

With regard to software, multiprocessing is simulated using threads, in total there are six threads being execute simultaneously, four Processor objetcs, a clock and a graphic interface.

***Palabras clave*—Multiprocessor, cache coherence, MOESI, write-through.**

I. Introducción

Los protocolos de coherencia de caché son necesarios en sistemas multiprocesador, pues en estos se permite que varias caché tengan copias simultáneas de una ubicación de memoria determinada. Esto abre paso a lo que se conoce como el problema de coherencia de caché, y puede ser descrito como el desafío de mantener sincronizadas varias caché locales, cuando uno de los procesadores actualiza su copia local ó de memoria, de datos que poseen otras caché [1]; esto con la intención de garantizar que el multiprocesamiento sea invisible para diversos clientes, como se muestra en la figura 1.

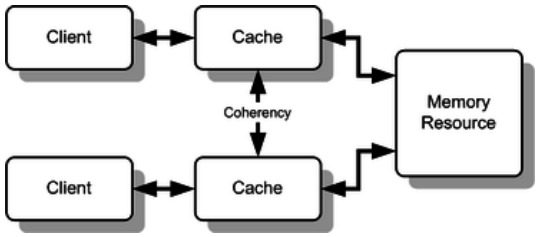

Figura 1: Problema de coherencia de caché.

De modo que estos protocolos permiten determinar a partir de estados, cuando es necesario realizar una lectura de memoria (o inclusive evitar estas) para actualizar un valor que contengan. Así, en este documento se describe el proceso de investigación, implementación y resultados de la simulación de un sistema multiprocesador que utiliza el protocolo MOESI, acompañado por una política de escritura de *write-through*. Además y dado que se cuenta con una única memoria, se debe de controlar que no exista un acceso concurrente a la misma, esto a través de un bus que se encarga de aceptar o rechazar estas peticiones.

Finalmente, en este documento se describe el marco teórico en que se fundamenta el proyecto, una explicación detallada del sistema desarrollado, los resultados del mismo y una sección de conclusiones que surgen del análisis de los resultados.

II. Marco teórico

Como se mencionó, el problema a tratar es el de mantener coherencia en las caché de diferentes procesadores, en un sistema multiprocesador. De modo que si existieran dos procesadores y cada una de sus caché locales contaran con el mismo dato, cada una tenga la forma de garantizar el acceso a la versión más actualizada de dicho dato, aún si ha sido modificado recientemente. El caso contrario a lo descrito se muestra en la figura 2, en que la caché 1, actualiza el dato de la dirección *0001* y la caché 2, aunque también cuenta con una copia del dato, no es capaz de detectar que lo debe actualizar.

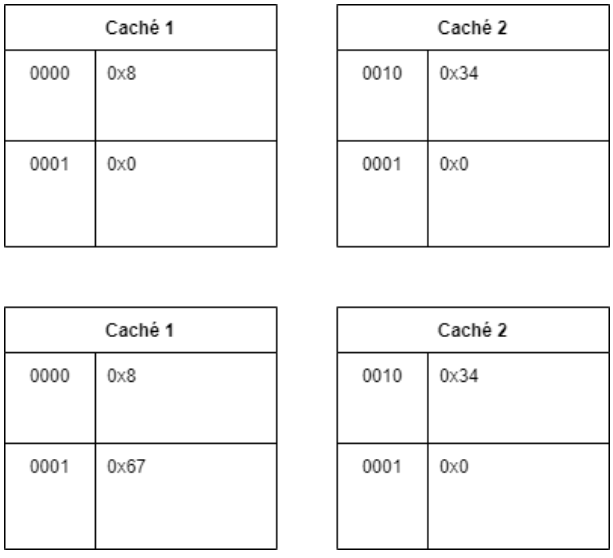

Figura 2: Dos caché sin consistencia en sus datos.

Para evitar el problema descrito anteriormente, se puede utilizar el protocolo de coherencia de caché MOESI, el mismo cuenta con los siguientes estados, según [2]:

- **Modified.** Indica que la línea de caché está sucia, es decir, su valor es diferente al de la memoria principal.
- **Owned.** Una línea de caché en este estado contiene la copia correcta más reciente de los datos. Solo un procesador puede mantener los datos en estado de propiedad, mientras que todos los demás procesadores deben mantener los datos en estado compartido.
- **Exclusive.** La línea de caché está presente solo en la caché actual y su valor coincide con el valor de la memoria principal.
- **Shared.** Este bloque no está modificado y está presente en al menos otra caché. Los datos se pueden leer sin necesidad de reescribirlos en memoria.
- **Invalid.** Este bloque no es válido y debe obtenerse el valor actualizado de la memoria o de otra caché.

Dicho protocolo permite las transiciones que se muestran en la figura 3. El cual puede entenderse mejor viendo el ejemplo de la figura 4, para el que existen tres procesadores. Cuando *P1* realiza una lectura de la posición de memoria *X*, guarda esta en su caché con estados *E*, seguidamente *P2* también lee *X* por lo que ahora ambos tienen el estado *S*. Finalmente, cuando el procesador *P2* realiza una modificación, adquiere el estado *M* y *P1* pasa a *I*, luego de que *P1* realice la lectura del nuevo valor de *X*; *P1* para a *S* y *P2* a *O*. De este modo se logra ver como se conserva la coherencia del dato *X* para ambas caché.

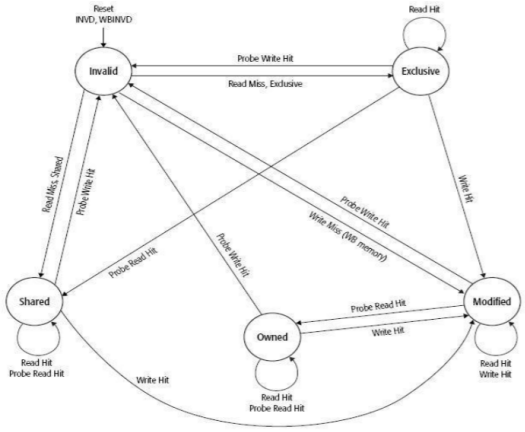

Figura 3: Transiciones permitidas en el protocolo MOESI [3].

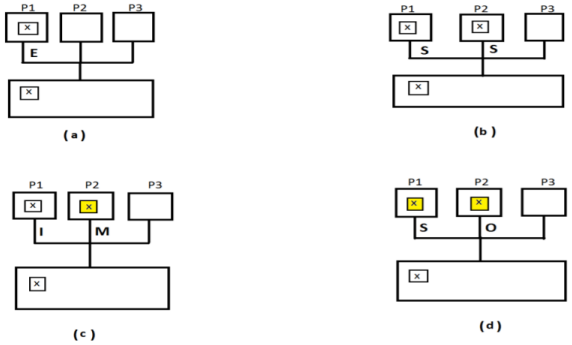

Figura 4: Ejemplo del protocolo MOESI [3].

Ahora, cuando un procesador modifica los datos en su caché, se debe definir cuándo la CPU hace que los datos modificados se coloquen en la memoria principal, para esto hay dos enfoques conocidos como *write-trought* y *write-back*. En este proyecto se implementa *write-trought*, de modo que siempre que la CPU modifica algún dato en la caché, se escribe inmediatamente la nueva información en las ubicaciones correspondientes de la memoria principal. Esto garantiza al sistema que cualquier dispositivo que acceda a la memoria principal, tendrá acceso exactamente a los mismos datos de las caché [4], sin embargo el uso del bus es mucho mayor.

III. Sistema desarrollado

En el sistema de simulación, los estados de las líneas de caché son actualizados a través de una comunicación entre procesadores, utilizando un sistema de monitoreo (*snoopy*) dentro de los procesadores, como se ve en la figura 5.

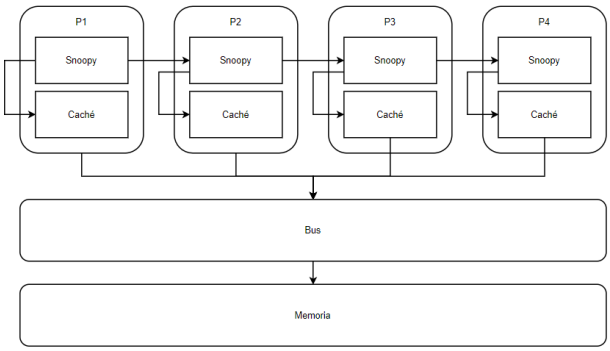

Figura 5: Diagrama del sistema implementado.

Además, cada uno de estos procesadores es capaz de ejecutar tres tipos de instrucciones, los cuales se describen a continuación:

- **Calc.** Esta instrucción realiza un calculo local en el procesador y tarda un ciclo.
- **Read.** Realiza una lectura de una dirección de memoria, ya sea en la caché del procesador que lanza la instrucción, otra caché que contenga el dato ó en última instancia en memoria. Tarda un mínimo de dos ciclos de procesador.
- **Write.** Realiza una escritura de una dirección de memoria en la caché del procesador que lanza la instrucción, además escribe el valor en memoria y dispara una rutina de escritura en el resto de cachés que también cuentan con el dato escrito.

Finalmente, el uso de *write-trought* como política de escritura, implica que se puede prescindir de los estados de *M* e *I* pues la memoria y las caché están siempre actualizadas. Esto también quiere decir que, el estado de *O* debe de asignarse a alguna caché en el mismo instante en que se detecta otra lectura o escritura de la etiqueta en cuestión, sin embargo como se mencionó anteriormente, los costo de comunicación son mucho mayores. De este modo las transiciones permitidas en el sistema se muestran en la figura 6.

IV. Resultados y análisis

Para esta sección se cargaron instrucciones bien sabidas en los procesadores, con la intención de que se

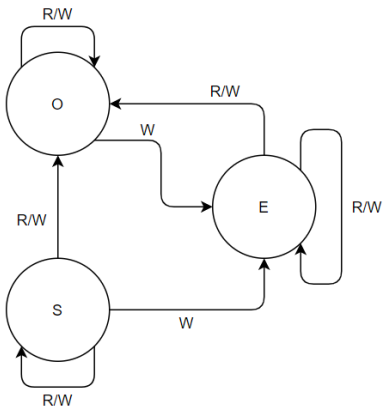

Figura 6: Transiciones permitidas en el protocolo implementado.

lograran comprobar la mayor cantidad de transiciones, sin embargo estas se ejecutan aleatoriamente, según cada procesador logre acceder a los recursos compartidos (bus y memoria). Dando como resultado el programa que se muestra a continuación y que sigue el formato Ciclo/Procesador/Instrucción/Dirección/Dato obtenido o a escribir:

```
Ciclo 2, Proc: 2, Calc
Ciclo 2, Proc: 4, Read mem: 0 0x0
Ciclo 3, Proc: 2, Calc
Ciclo 4, Proc: 2, Calc
Ciclo 4, Proc: 1, Read cache: 0 0x0
Ciclo 4, Proc: 4, Calc
Ciclo 5, Proc: 1, Calc
Ciclo 5, Proc: 3, Read mem: 1 0x0
Ciclo 7, Proc: 3, Calc
Ciclo 8, Proc: 3, Calc
Ciclo 8, Proc: 1, Write 100 0x5
Ciclo 11, Proc: 1, Write 10 0x3
New instruction inserted: Read, 101
Ciclo 14, Proc: 1, Read mem: 101 0x0
Ciclo 16, Proc: 3, Read cache: 0 0x0
Ciclo 17, Proc: 1, Read my cache: 0 0x0
Ciclo 18, Proc: 2, Read cache: 0 0x0
Ciclo 19, Proc: 3, Read my cache: 1 0x0
Ciclo 20, Proc: 3, Calc
Ciclo 20, Proc: 2, Calc
Ciclo 20, Proc: 1, Read my cache: 0 0x0
Ciclo 21, Proc: 1, Calc
Ciclo 21, Proc: 2, Calc
Ciclo 21, Proc: 3, Calc
Ciclo 21, Proc: 4, Write: 100 0x11
Ciclo 22, Proc: 2, Calc
Ciclo 24, Proc: 2, Read my cache: 0 0x0
Ciclo 25, Proc: 4, Write: 10 0xC
Ciclo 26, Proc: 2, Calc
Ciclo 27, Proc: 2, Calc
Ciclo 28, Proc: 2, Calc
Ciclo 28, Proc: 4, Write: 1 0x8
Ciclo 31, Proc: 2, Read my cache: 0 0x0
Ciclo 31, Proc: 4, Read cache: 0 0x0
Ciclo 32, Proc: 3, Read my cache: 0 0x0
Ciclo 33, Proc: 4, Read my cache: 0 0x0
Ciclo 33, Proc: 2, Calc
Ciclo 34, Proc: 4, Calc
Ciclo 34, Proc: 2, Calc
Ciclo 34, Proc: 3, Read my cache: 1 0x8
Ciclo 35, Proc: 2, Calc
Ciclo 35, Proc: 3, Calc
Ciclo 35, Proc: 1, Write: 100, 0x5
Ciclo 36, Proc: 3, Calc
Ciclo 38, Proc: 1, Write: 10 0x3
Ciclo 41, Proc: 4, Write: 100 0x11
Ciclo 44, Proc: 4, Write: 10 0xC
Ciclo 47, Proc: 4, Write: 1 0x8
Ciclo 50, Proc: 4, Read cache: 0 0x0
Ciclo 51, Proc: 2, Read my cache: 0 0x0
```

Así, en la figura 7 se muestra la ejecución hasta el cuarto ciclo, en que se realizó una lectura de la dirección *0x0* en el procesador 4 y luego en el 1, razón por la que el procesador 1 adquiere estado de *O*.

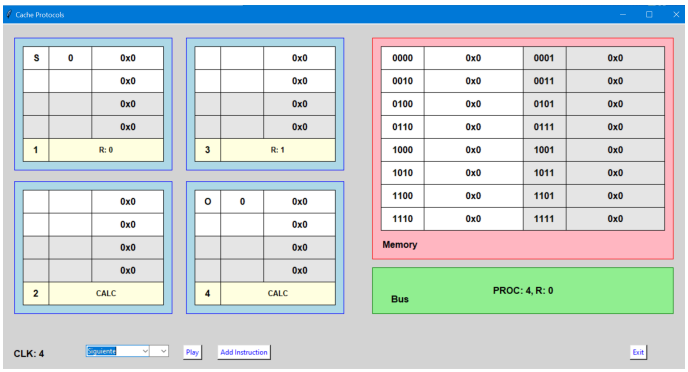

Figura 7: Ejecución del programa hasta el ciclo 4.

En el ciclo 11 de ejecución (figura 8) se ve una lectura del procesador 3 a la dirección *1*, además obtiene el estado *E*. Luego de una escritura del procesador 1 en la dirección *100* y antes de la escritura en *10*, se carga una nueva instrucción de lectura en *101*. En la figura 9 se muestra el resultado de la ejecución de esta instrucción nueva (que se carga en el procesador 1 según la configuración del usuario) y de la escritura que había quedado pendiente, además de una lectura del procesador 3 en *0*.

Hasta el ciclo 24 únicamente suceden operaciones de lectura y una escritura del procesador 4 en *100*, como se muestra en la figura 10. Esta escritura es importante pues ocasiona que la escritura del procesador 4 en *10* que inicia en el ciclo 25, reemplace la línea de *0*, pasando así el estado de *O* al procesador 3 (esto aleatoriamente), este comportamiento es ilustrado por la figura 11.

En el ciclo 35 y luego de varias operaciones de lectura, se inicia una escritura del procesador 1 en *100* (figura 12), que al reemplazar la línea de caché que contenía a *10*, altera el estado de la misma línea en el procesador 4, haciendo que cambie a *E*, el resultado de esto se ve en la figura 13.

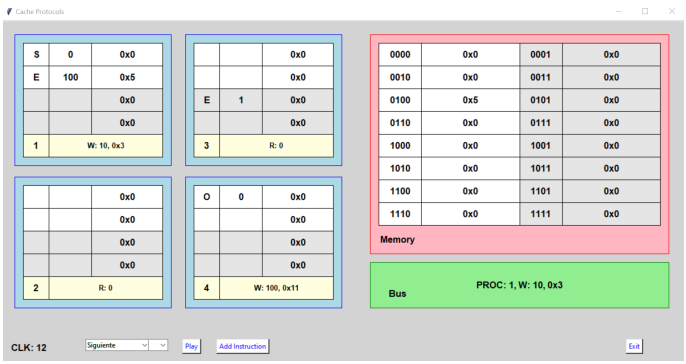

Figura 8: Ejecución del programa hasta el ciclo 11.

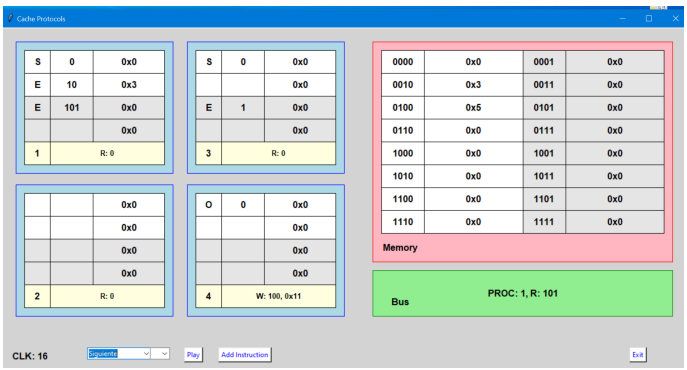

Figura 9: Ejecución del programa hasta el ciclo 16.

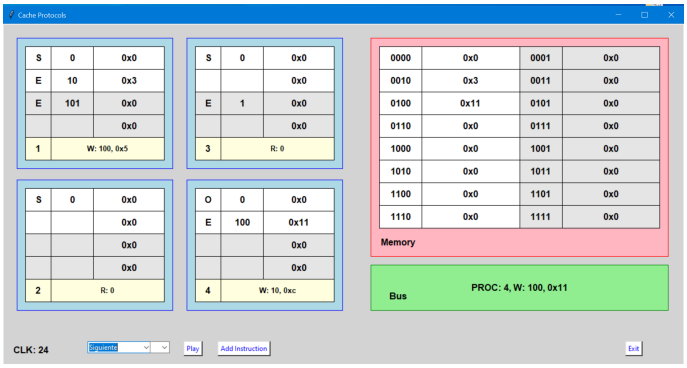

Figura 10: Ejecución del programa hasta el ciclo 24.

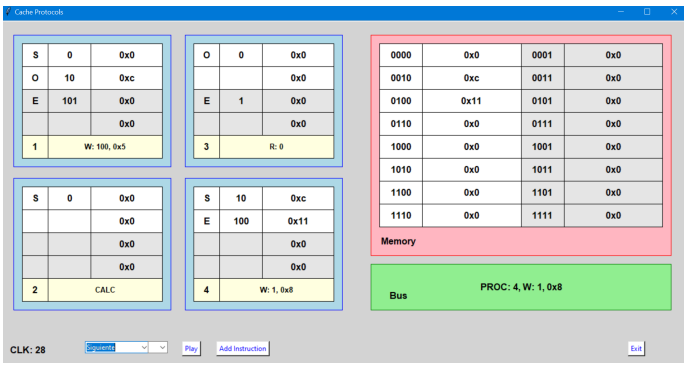

Figura 11: Ejecución del programa hasta el ciclo 28.

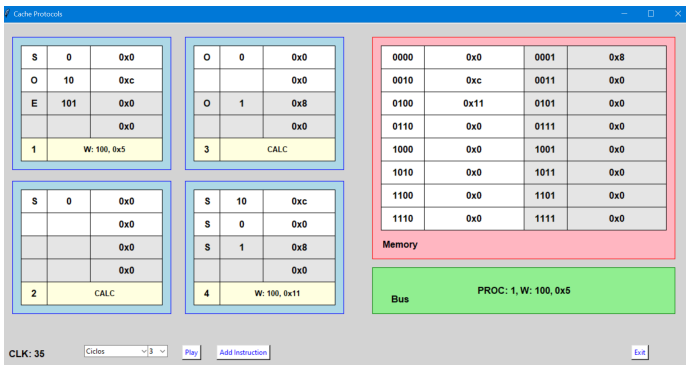

Figura 12: Ejecución del programa hasta el ciclo 35.

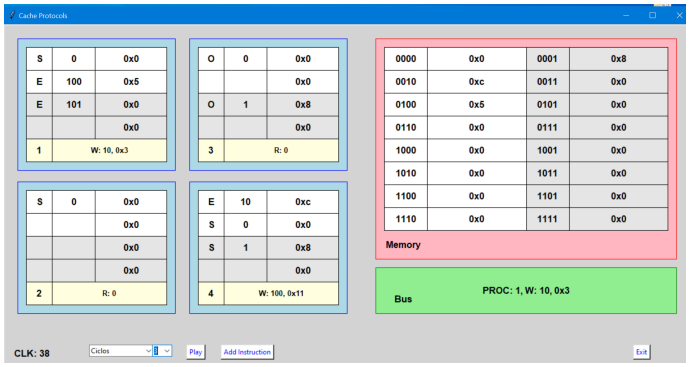

Figura 13: Ejecución del programa hasta el ciclo 38.

V. Conclusiones

Así, se logra exhibir la importancia de garantizar la coherencia en las caché en sistemas multiprocesador, de modo que el objetivo en última instancia, es lograr que está paralelización resulte invisible para el usuario final. Eso se puede alcanzar utilizando protocolos basados en estados de las líneas de caché como lo es *MOESI*, que acompañado de una política de escritura *write-trought* logra simplificar bastante la implementación, sin embargo el costo de la comunicación entre procesadores es bastante importante. Esto a nivel de la simulación ocasiona la necesidad de un *clock* lento, pues se necesita permitir que el tiempo sea suficiente para que los procesadores y *snoopy tags* realicen su labor, mientras que en una implementación en hardware se requeriría de mayor consumo energético y un uso intensivo del bus.

Referencias

[1] Seralahthan. (2019, Mar 23). *Cache Coherence Problem and Approaches*. [Online]. Recuperado de: https://medium.com/@TechExpertise/cache-coherence-problem-and-approaches-a18cdd48ee0e

[2] GeekForGeeks. (2019, Oct 17). *Cache Coherence Protocols in Multiprocessor System*. [Online]. Recuperado de: https://www.geeksforgeeks.org/cache-coherence-protocols-in-multiprocessor-system/

[3] S. Dey, & M. Nair. (2014). *Design and implementation of a simple cache simulator in Java to investigate MESI and MOESI coherency protocols*. International Journal of Computer Applications, 87(11).

[4] S. Raman. (1996). U.S. Patent No. 5,555,398. Washington, DC: U.S. Patent and Trademark Office.