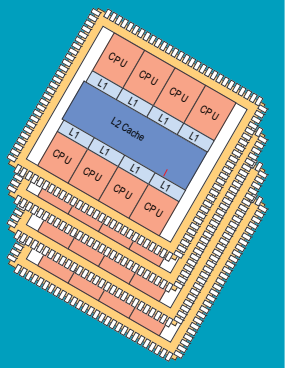
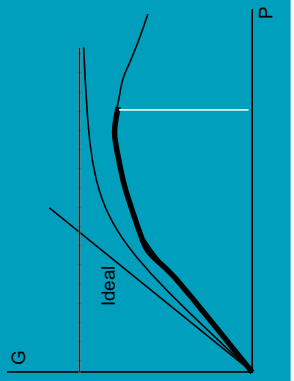
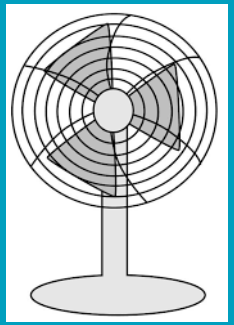






Multiprocesadores

$$P = C_e \cdot v^2 \cdot f + v \cdot I_f$$

J.M. Llabería

Contenido

Capítulo 3
Consistencia de memoria y coherencia de cache

Los multiprocesadores han sido introducidos después de una amplia utilización de sistemas uniprocesador. Estos últimos se han utilizado para ejecutar programas serie. También mediante la multiplexación del procesador, denominada usualmente tiempo compartido, un sistema uniprocesador se ha utilizado para ejecutar varios procesos serie concurrentemente o varios procesos o hilos pertenecientes al mismo programa paralelo.

Un multiprocesador tiene como objetivos reducir el tiempo de ejecución de una aplicación paralela y/o incrementar la productividad al ejecutar varias tareas serie. Esto es, el número de tareas serie procesadas por unidad de tiempo.

Organización. En la Figura 3.1 se muestran organizaciones básicas de sistemas multiprocesador. Estas organizaciones pueden considerarse una extensión o transición natural de un sistema uniprocesador (máquina von-Neumman), donde mediante una red de interconexión se conectan varios procesadores y módulos de memoria.

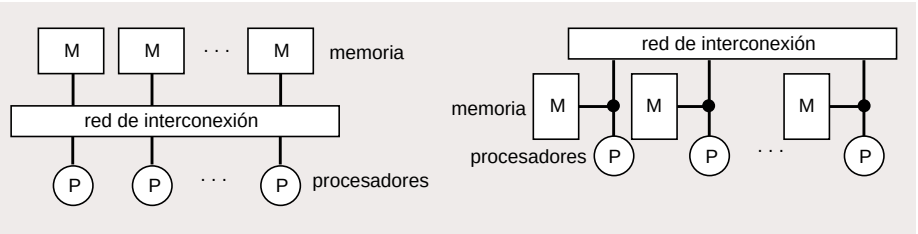


Figura 3.1 Multiprocesador de memoria compartida. a) Multiprocesador con memoria en un lado de la red de interconexión y procesadores en el otro lado y b) Multiprocesador con memoria físicamente distribuida.

En el multiprocesador mostrado en la parte izquierda de la Figura 3.1, en un lado de la red de interconexión se ubican procesadores y en el otro lado de la red se ubican módulos de memoria. En este tipo de multiprocesador la memoria podría estar constituida por un único módulo dependiendo de la red de interconexión utilizada. En el caso de que se utilicen varios módulos se puede acceder a cada uno de ellos de forma concurrente.

En el multiprocesador de la parte derecha de la Figura 3.1 los nodos son parejas módulo de memoria y procesador. En esta organización, notemos que uno de los módulo de memoria está más cerca de un procesador que los otros módulos de memoria y por tanto, la latencia de acceso es menor.

Espacio de direcciones global. Un sistema multiprocesador con memoria compartida dispone de un espacio de direcciones físicas global al que se tiene acceso desde cualquier procesador. El acceso al espacio de direcciones global se efectúa mediante instrucciones de load y store. Por ejemplo, esta característica es útil para los procesos del sistema operativo que acceden a las mismas estructuras de datos (estructuras de datos compartidas) y también facilita que los procesos puedan migrar entre procesadores. Por otro lado simplifica tareas tales como la distribución de la carga de trabajo entre los procesadores.

Programación. Los programas tanto serie como paralelos utilizan instrucciones store y load sobre una posición de memoria para comunicar información entre instrucciones distintas del mismo proceso o entre procesos. En este último caso, las instrucciones load y store se utilizan también para establecer ordenación entre dos procesos o hilos que se ejecutan concurrentemente, pero que cooperan para realizar una misma tarea. Como ejemplo, en la Figura 3.2 se muestra una sincronización punto a punto entre dos hilos. La variable aviso se inicializa a cero y permite sincronizar la lectura de la variable A en el hilo H2 con la escritura de la misma variable en el hilo H1.

Hilo H1	Hilo H2
A = 2.36	While (aviso <>1) {}
aviso = 1	T = A

Figura 3.2 Sincronización punto a punto entre dos hilos utilizando eventos. El valor inicial de la variable aviso es cero.

Un programador intuitivamente observa las operaciones de memoria (una en cada instante) en el orden especificado en el programa. Dada una posición de memoria, un load devuelve el último valor escrito en la posición de memoria accedida. Una instrucción store determina el valor que será devuelto por una instrucción load posterior o más joven hasta que la siguiente instrucción store actualice la posición de memoria.

En el trozo de código mostrado la Figura 3.2 se utiliza la variable aviso para establecer un orden entre la escritura de la variable A en el hilo H1 con la lectura de la misma variable en el hilo H2. Mediante la asignación del valor uno, a la variable aviso en el hilo H1, estamos indicando el instante a partir del cual se puede leer la variable A en el hilo H2. Mientras la variable aviso sea distinta de uno, el hilo H2 ejecuta un bucle, cuyo cuerpo es una instrucción load que lee la variable aviso. El programador espera que el hilo H2 lea el valor 2.36 al acceder a la variable A.

En programas paralelos que se ejecutan en un multiprocesador, la idea es aplicar razonamientos conocidos por diseñadores de sistemas operativos y bases de datos en un uniprocesador con tiempo compartido. Por tanto, interesa que la semántica del espacio de direcciones global, de un sistema multiprocesador, ofrezca un modelo de programación (visión de la memoria) compatible con los sistemas uniprocesador con tiempo compartido.

Semántica del espacio de direcciones global (consistencia de memoria).

En un procesador que se multiplexa o en un multiprocesador, entendemos por semántica del espacio de direcciones global compartido, cómo los procesos o hilos observan el entrelazado de los accesos a memoria, efectuado por cada uno de ellos de forma autónoma. Esta semántica es la que se utiliza al desarrollar programas paralelos y es la que permite razonar y predecir el comportamiento de los programas. Por tanto, hay que conocer la semántica del espacio de direcciones.

En la Figura 3.3 se muestra mediante un esquema la relación entre algoritmo, modelo abstracto de memoria global y acceso a posiciones de memoria en una máquina de cómputo. El modelo abstracto de memoria es una especificación formal de cómo el sistema de memoria es observado por el programador. Esto elimina la posible brecha que puede existir entre el comportamiento esperado del programa y cómo se comporta el sistema. En concreto, el modelo de memoria impone restricciones sobre el valor que puede ser devuelto en una lectura al ejecutar un programa.

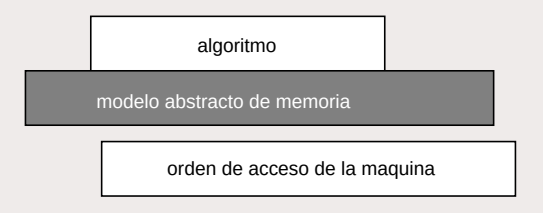


Figura 3.3 Modelo abstracto de memoria global. Esquema de relaciones entre niveles.

El modelo abstracto de memoria o modelo de consistencia de memoria influye en todo el sistema de cómputo, desde la programación, pasando por el rendimiento y finalizando por la portabilidad. El programador utiliza el modelo de consistencia para razonar acerca de la corrección de los programas. Los arquitectos y diseñadores de compiladores desarrollan optimizaciones que se explotan a nivel hardware y software, con el objetivo de incrementar el rendimiento. La portabilidad indica si un software desarrollado para un sistema de cómputo se puede ejecutar en otro sistemas de cómputo.

Consistencia de memoria. Determina cuándo un valor escrito en una posición de memoria será devuelto en una lectura.

La especificación del lenguaje máquina de un procesador incluye la semántica del espacio de direcciones global compartido o modelo de consistencia de memoria.

Organización con jerarquía de memoria. En la Figura 3.4 se muestran las organizaciones mostradas en la Figura 3.1 con cache privadas. La función de la cache es reducir el tiempo medio de acceso a instrucciones y datos por parte de cada procesador y reducir el ancho de banda demandado por cada procesador en la red de interconexión y en la memoria.

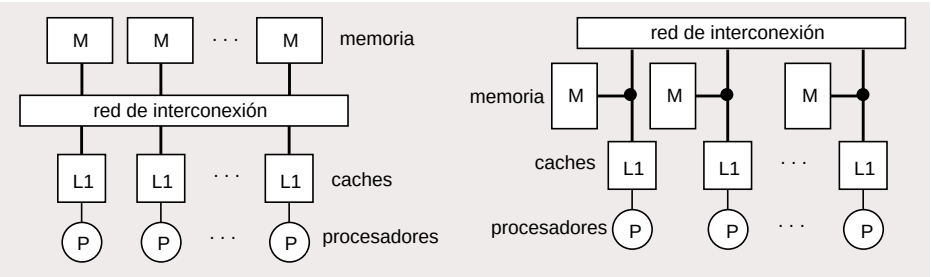


Figura 3.4 Organizaciones de sistemas multiprocesador con cache privadas.

Además, las caches privadas en los multiprocesadores mostrados en la Figura 3.4, habilitan la posibilidad de que existan copias de un mismo bloque de memoria en cada una de ellas, y por tanto, aparece la necesidad de que estas copias sean coherentes.

Coherencia de cache. Determina qué valor puede ser devuelto en una lectura a una posición de memoria.

El mecanismo utilizado en un multiprocesador para mantener la coherencia es transparente al lenguaje máquina. Debe ser eficiente, con coste reducido y no intrusivo.

Podemos decir que:

- Consistencia de memoria: está relacionada con el orden de todas las lecturas y escrituras a todas las posiciones de memoria.
- Coherencia de cache: está relacionada con el orden de todas las lecturas y escrituras a cada posición de memoria individual.

En este capítulo, en primer lugar se desarrolla el modelo de consistencia secuencial de memoria. Posteriormente se analiza como mantener coherentes las caches privadas (coherencia de cache).

Finalmente se muestra que el modelo de consistencia secuencial de memoria establece unas restricciones que son más estrictas que las restricciones necesarias que espera un programador al ejecutar un programa paralelo. Por ello se han desarrollado modelos de consistencia de memoria relajados, cuyo objetivo es posibilitar diseños hardware que incrementen el rendimiento y habilitar optimizaciones del compilador, que en caso contrario deben inhibirse. En contrapartida el programador debe prestar mayor atención al programar. En concreto, debe especificar en el código las zonas donde debe respetarse consistencia secuencial.

CONSISTENCIA SECUENCIAL DE MEMORIA

Recordemos que, al escribir programas paralelos que se ejecutan en un multiprocesador, la idea es aplicar razonamientos conocidos por diseñadores de sistemas operativos y bases de datos en un uniprocesador con tiempo compartido.

En la parte izquierda de la Figura 3.5 se muestra un modelo simplificado del proceso de multiplexación de un procesador. En este ejemplo, un procesador se multiplexa para ejecutar concurrentemente dos procesos serie. Un conmutador conecta un proceso a la memoria global y la posición del conmutador se determina de forma aleatoria después de cada acceso. Cada proceso efectúa los accesos a memoria en el orden especificado por el programador y el conmutador determina la serialización global entre todos los accesos. La multiplexación del procesador se produce entre acciones atómicas. En particular, las instrucciones load y store a una posición de memoria son operaciones (acciones) atómicas.

Informalmente, al ejecutar una programa serie o paralelo en un uniprocesador, que se multiplexa, esperamos que una lectura de una posición de memoria devuelva el valor más reciente escrito en esta posición de memoria.

Esto es, el valor escrito en una posición de memoria es observado por el siguiente load (del mismo o distinto hilo) que accede a esa posición de memoria, si antes no se ha escrito otra vez en esa posición de memoria.

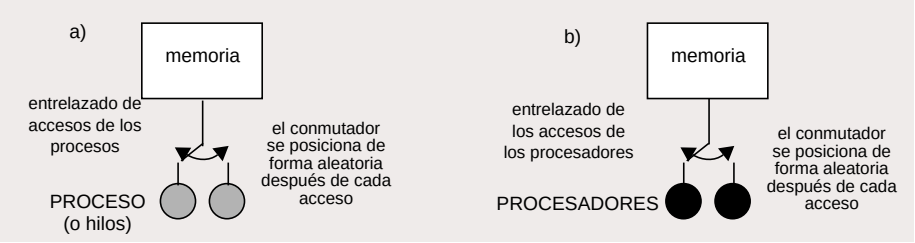


Figura 3.5 Modelo de consistencia de memoria. a) sistema uniprocesador con tiempo compartido y b) sistema multiprocesador.

El modelo de consistencia de memoria en un procesador con tiempo compartido se denomina consistencia secuencial. Informalmente, disponer de consistencia secuencial en un multiprocesador requiere que la ejecución de un programa paralelo se comporte igual que una ejecución entrelazada de los hilos del programa paralelo en un uniprocesador con tiempo compartido.

Conceptualmente, un multiprocesador con consistencia secuencial se comporta como si todos los procesadores tuvieran un turno para acceder a memoria, creando un flujo entrelazado de accesos, en el que los accesos de cada procesador individual están ordenados (Figura 3.5 parte derecha). Los procesadores efectúan accesos a memoria uno cada vez y el flujo de accesos a memoria es el entrelazado de accesos de los procesadores. Esto es, los accesos a memoria de todos los procesadores son atómicos y se puede construir un orden serie del entrelazado de accesos a memoria.

En la parte izquierda de la Figura 3.6 se muestra un modelo simplificado de multiprocesador. En la parte derecha de la figura se muestra un entrelazado de accesos cuando se ejecutan los dos hilos mostrados en la parte izquierda. En este entrelazado de accesos, se puede identificar el orden en el cual se han especificado los accesos en cada hilo. Por otro lado, una instrucción load lee el valor establecido por la instrucción store previa en el entrelazado.

Consistencia secuencial [Lamport]. Un multiprocesador es secuencialmente consistente si, el resultado de cualquier ejecución es el mismo que se obtendría cuando las operaciones de todos los procesadores se hubieran ejecutado en un orden serie (una cada vez), y las operaciones de cada uno de los procesadores aparecen en dicha secuencia en el orden especificado por el programador[1].

1. Desde el punto de vista del programador es un multiprocesador sin caches privadas y todos los accesos a posiciones de almacenamiento se encaminan a memoria, la cual sirve un acceso cada vez.

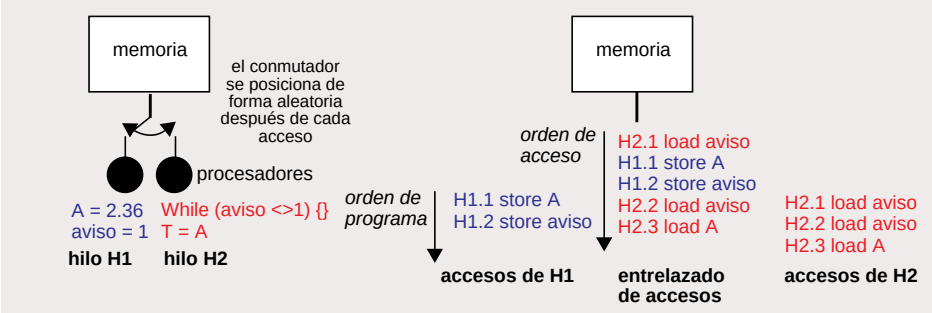


Figura 3.6 Entrelazado de accesos de dos hilos que se sincronizan. La variable aviso ha sido inicializada a cero. Notación: Hx.y, donde x es el número de hilo e y identifica el orden de programa.

De la definición previa de consistencia secuencial se extraen las siguientes condiciones.

Orden del programa. Dado un procesador, orden en el cual las operaciones de memoria o accesos a memoria son especificados por el programador (*las operaciones de cada uno de los procesadores aparecen en dicha secuencia en el orden especificado por el programador*).

Respetar el orden de programa indica que hay que mantener el orden de acceso en las cuatro combinaciones posibles de instrucciones load y store a posiciones de almacenamiento distintas (Figura 3.7)[2].

orden de programa				
↓	lectura	lectura	escritura	escritura
	lectura	escritura	lectura	escritura

Figura 3.7 Modelo de consistencia secuencial. Ordenes que hay que respetar en los accesos a posiciones de memoria.

Atomicidad. Todos los procesadores observan todos los accesos a memoria en el mismo orden (*cuando las operaciones de todos los procesadores se hubieran ejecutado en un orden serie, una cada vez*)[3].

Atomicidad indica una única operación indivisible o que se realiza instantaneamente.

2. Por otro lado, dado un hilo, para preservar su semántica, las dependencias al acceder a una posición de memoria deben respetarse.
3. Un acceso a memoria ha finalizado antes de que se efectúe el próximo acceso a memoria de cualquier procesador. En otras palabras, un acceso a memoria es visible instantáneamente a todos los procesadores.

Análisis del modelo de consistencia secuencial. En la Figura 3.8 se muestran dos hilos que efectúan dos accesos a memoria cada uno[4]. Para representar la acción del conmutador de la Figura 3.5 utilizamos un árbol binario[5]. Un orden de ejecución empieza en la raíz y finaliza en una hoja. Dada una rama, en cada nivel se representa la sentencia ejecutada, teniendo en cuenta que las sentencias se ejecutan en orden de programa y las sentencias que se han ejecutado previamente.

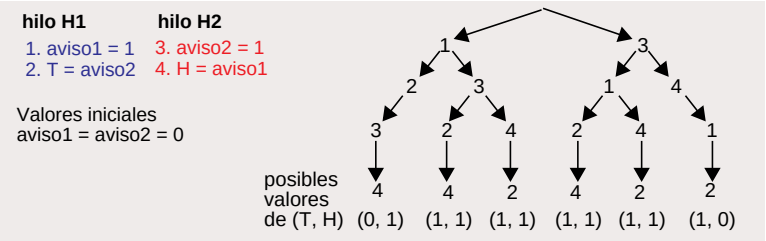


Figura 3.8 Posibles entrelazados secuencialmente consistentes.

Deducimos que hay seis posibles entrelazados que respetan el orden de programa y los posibles valores del par de variables (T, H) son (0,1) (1, 0) y (1, 1). Estos entrelazados son secuencialmente consistentes. Notemos que el valor del par (0, 0), al iniciar la ejecución de los hilos, no es posible observarlo al finalizar la ejecución. Un entrelazado que produzca el par (0, 0) no es secuencialmente consistente.

En una ejecución de un programa paralelo se observa un entrelazado. En ejecuciones distintas del programa paralelo pueden observarse entrelazados distintos.

Necesidades del modelo de consistencia secuencial

Orden de programa. Para garantizar orden de programa, un procesador debe recibir una respuesta del sistema de memoria.

- En una instrucción load la respuesta es el dato.
- En una instrucción store el sistema de memoria responde explícitamente, mediante una confirmación de la escritura[6].

Atomicidad. Un acceso a memoria es atómico (indivisible)[7]. Entonces, una vez una escritura llega a memoria, la posición de memoria accedida se actualiza y es visible por un load posterior, a la misma posición de memoria,

4. El código se corresponde con un esqueleto del algoritmo de Dekker para exclusión mutua.
5. En cualquiera de los dos esquemas mostrados en la Figura 3.5.
6. En un uniprocesador no es necesario una respuesta explícita en una instrucción store, ya que sólo hay un camino a memoria que mantiene el orden de los accesos a memoria. Además en este único camino se pueden gestionar dependencias de datos. Por tanto, en un uniprocesador al emitir una instrucción store no se espera respuesta del sistema de memoria.

efectuado por cualquier procesador, si previamente no se ha ejecutado otro store a la misma posición de memoria. Respecto a una instrucción load, el valor que se lee de memoria es el valor almacenado por la instrucción store previa, en el entrelazado de accesos de todos los procesadores, a esa posición de memoria.

- Paralelismo en memoria. En un sistema de memoria construido con un conjunto de módulos de memoria entrelazados también se cumple la condición de atomicidad[8]. Si los accesos son al mismo módulo de memoria, el orden está determinado por el instante en el que acceden al módulo de memoria. Accesos concurrentes a módulos de memoria distintos se efectúan de forma paralela y pueden ordenarse arbitrariamente, entre ellos, mientras se mantenga el orden en cada módulo de memoria.

En la Figura 3.9 se muestra un ejemplo donde el sistema de memoria dispone de dos módulos de memoria. La variable aviso1 está ubicada en el módulo M1 y la variable aviso2 en el módulo M2. En el primer instante de tiempo cada procesador accede a un módulo de memoria. Después de recibir la confirmación de las respectivas escrituras, cada procesador emite el siguiente acceso a memoria. Estos accesos son instrucciones load y también acceden a memoria de forma paralela. En la parte inferior derecha de la figura se muestran dos de los cuatro posibles entrelazados, secuencialmente consistentes, que pueden construirse a partir de los entrelazados parciales.

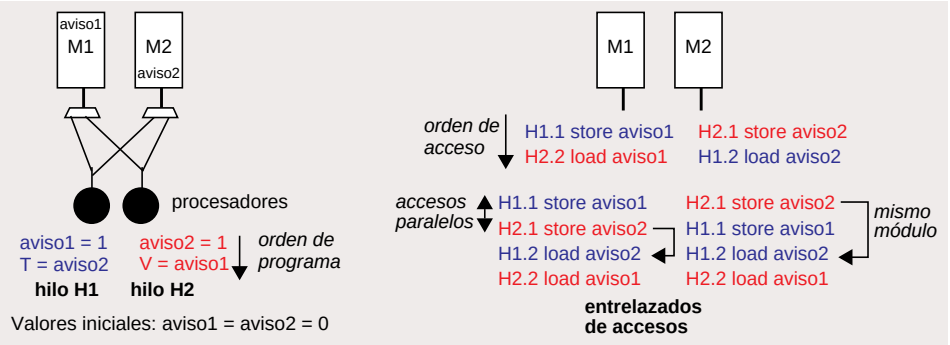


Figura 3.9 Sistema de memoria con dos módulos. Dos entrelazados posibles de accesos. Notación: Hx.y, donde x es el número de hilo e y identifica el orden de programa.

7. El acceso consolida con respecto a todos los procesadores antes de que el próximo acceso, en el entrelazado global, se inicie.
8. Estamos suponiendo orden de programa. Entonces, en un procesador el siguiente acceso a memoria se inicia después de que haya consolidado el previo. Por tanto, garantiza que los accesos de un procesador, a módulos de memoria distintos, se efectúan en orden de programa. También garantiza que una red con encaminamiento adaptativo (no siempre se utiliza el mismo camino para acceder desde un procesador a un módulo de memoria) no desordena, respecto al orden de programa, accesos efectuados por un procesador al mismo módulo de memoria.

Una propiedad que se puede extraer, de un sistema con varios módulos de memoria, es que la atomicidad de las escrituras[9] no requiere una serialización estricta de todos las escrituras. Es suficiente con la atomicidad de las escrituras a cada posición de memoria[10]. A partir de estos órdenes parciales se puede construir un orden global secuencialmente consistente.

Orden de programa

El orden, en que el programador ha especificado los accesos a memoria en un hilo, debe ser observado por todos los procesadores. Ningún componente del multiprocesador tiene que modificar el orden, ya sea el compilador, el procesador, la red de interconexión o memoria. Para ello hay que utilizar mecanismos explícitos o implícitos que permitan garantizar el orden de programa.

Implicaciones en compilación

En la parte izquierda de la Figura 3.6 se muestra un modelo simplificado de multiprocesador. En la parte derecha se muestra un entrelazado de accesos cuando se ejecutan los dos hilos mostrados en la parte izquierda.

El objetivo del código es comunicar el valor de A en el hilo H1 al hilo H2. La forma de establecer una relación de orden entre una escritura y una lectura, efectuadas por procesadores distintos, es utilizar una sincronización mediante eventos. Para ello se utiliza más de una posición de almacenamiento. Para preservar el orden, entre accesos a la misma posición de memoria, efectuados por procesadores distintos, hay que respetar el orden entre accesos a posiciones de almacenamiento distintas, efectuados por el mismo procesador[11].

En el ejemplo de la Figura 3.6 es importante el orden de programa, aunque si los dos hilos se analizan por separado no se puede observar esta necesidad. En el hilo H1, entre la asignación de la variable A y la asignación de la variable aviso no existe ninguna dependencia de datos. Igualmente ocurre en el hilo H2 entre la lectura de la variable aviso y la lectura de la variable A. Ahora bien, al considerar el programa paralelo, se observa que mediante la variable aviso se establece una dependencia de control dentro de cada hilo (Figura 3.10).

En la Figura 3.10 se muestra mediante relaciones de orden qué espera el programador al ejecutar el programa. El programador espera que se respete el orden de programa, aunque no existen dependencias de datos dentro de cada

9. Orden global de todos los stores a todas las posiciones de memoria.
10. Notemos que un módulo de memoria puede ser una única posición de almacenamiento.
11. Notemos que esta restricción también se aplica en el caso de un procesador que se multiplexa.

hilo. Podemos decir que son dependencias de control dentro de cada hilo en particular y en general en el programa paralelo. El orden de programa determina a -> b y c -> d. La ordenación b -> c implica la ordenación a -> d, donde -> indica relación de precedencia.

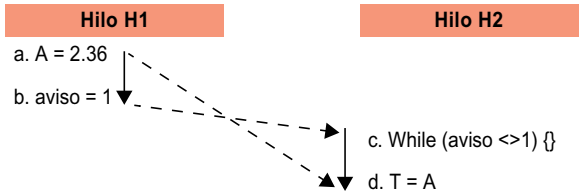


Figura 3.10 Sincronización punto a punto. Relaciones de dependencia entre dos hilos. El valor inicial de la variable aviso es cero.

El compilador no debe de utilizar algunas optimizaciones que se aplican en programas serie. Por ejemplo, el compilador no debe reordenar el orden de los accesos a memoria especificado por el programador[12]. Si en el ejemplo de la Figura 3.10 el compilador modifica el orden de acceso a las variables A y aviso, en cualquiera de los dos hilos, el programa no se comporta como esperamos.

Red de interconexión

La red de interconexión es un elemento que puede modificar la observación del orden de los accesos efectuados por un procesador por parte de otro procesador. En la Figura 3.11 se muestra un sistema multiprocesador que utiliza una red en malla. Cada nodo de la red tiene tres elementos: a) procesador, b) memoria y c) un encaminador, que encamina los accesos a otros nodos si no se sirven en este nodo. La memoria está entrelazada utilizando los bits menos significativos de la dirección. En el módulo de memoria del nodo 0 se almacena el bloque de memoria cero, en el nodo 1 se almacena el bloque de memoria uno y así sucesivamente.

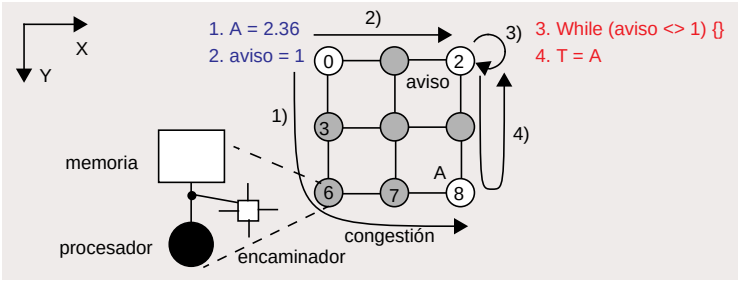


Figura 3.11 Multiprocesador que utiliza una red en malla.

12. En un apartado posterior se expone la necesidad de disponer de una primitiva específica para esta funcionalidad. Esta primitiva se utiliza cuando el modelo de consistencia de un lenguaje no es el modelo de consistencia secuencial.

En esta red, un acceso de un procesador a la memoria ubicada en otro nodo se encamina primero en la dirección Y y después en la dirección X. Por ejemplo, un acceso del procesador P0 a la memoria del nodo 8 se encamina en primer lugar por los nodos 3 y 6 y posteriormente por el nodo 7. En estas condiciones, los accesos entre dos nodos cualesquiera se transmiten de forma ordenada. Esto es, se mantiene orden punto a punto.

En el ejemplo de la Figura 3.11 el procesador P0 ejecuta un trozo de código de un hilo y el procesador P2 ejecuta otro trozo de código de otro hilo, los cuales pertenecen a un programa paralelo (Figura 3.10). Las variables A y aviso están almacenadas respectivamente en la memoria de los nodos 8 y 2.

Supongamos que se están transmitiendo por la red accesos de varios procesadores. Por congestión entendemos que el acceso de uno de ellos se ve retrasado por los accesos de los otros procesadores. Por ejemplo, en la Figura 3.11 existe mucha comunicación entre los nodos 3, 6 y 7. Entonces, supondremos que los accesos del procesador P0 al módulo de memoria ubicado en el nodo 8 experimentan retrasos.

Seguidamente se muestra la necesidad de que memoria confirme un acceso de escritura. El objetivo de esta señal de confirmación es garantizar que los accesos a memoria de un procesador sean observados por los otros procesadores en orden de programa.

No se confirma una escritura. Un procesador, después de emitir una instrucción store, ejecuta la siguiente instrucción sin esperar una confirmación. El procesador P0 emite una escritura que accede a la memoria del nodo 8. Seguidamente emite la siguiente escritura que accede a la memoria del nodo 2. El segundo store actualiza la variable aviso. El camino que siguen los dos accesos en la red de interconexión es distinto y también lo es el retardo en llegar al módulo de memoria

En la Figura 3.12.a se muestra un diagrama temporal del instante de inicio de una operación en un procesador, el instante en que se accede a memoria y en el caso de un load, el instante en el cual el procesador dispone del dato. En este último caso se utiliza una línea quebrada que parte del procesador, llega al módulo de memoria y vuelve al procesador. En el caso de una instrucción store la línea finaliza en el módulo de memoria.

En el procesador P2, al ejecutar el bucle se determina que la variable aviso ha tomado como valor uno (3). Por tanto, se lee la variable A (4). En el camino desde el nodo cero al nodo 8 hay congestión. El store emitido por el procesador P0 aún no ha actualizado memoria cuando se lee la variable A desde el

procesador P2. Entonces, el load del procesador P2 no lee el valor 2.36 sino un valor previo. Por tanto, la ejecución no cumple el modelo de consistencia secuencial.

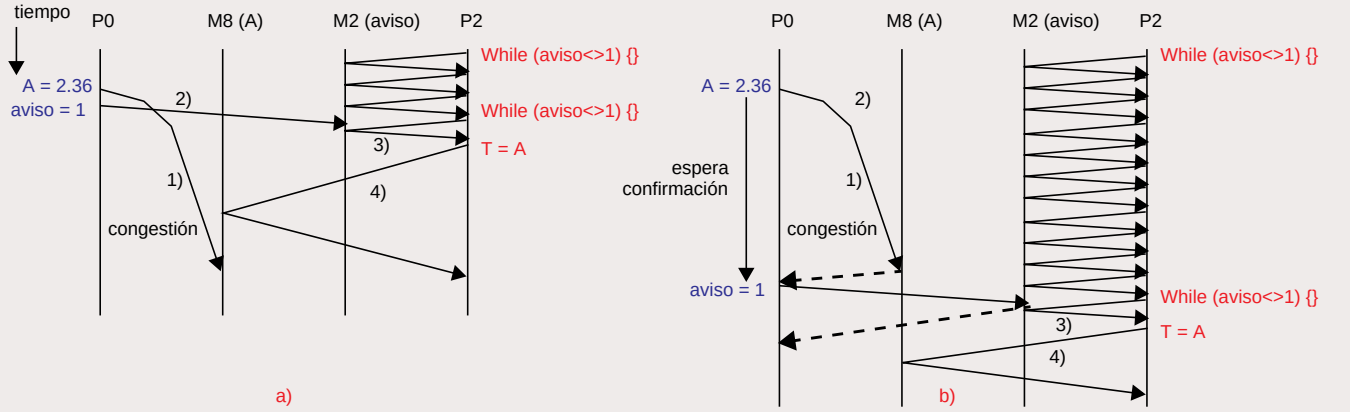


Figura 3.12 Diagrama temporal de los accesos mostrados en la Figura 3.11: a) No se confirman las escrituras, b) Se confirman las escrituras. Una línea de trazo discontinuo grueso es la confirmación de una escritura. La numeración de las acciones se corresponde con la numeración de la Figura 3.11.

El procesador P0 ha iniciado los accesos a memoria en secuencia, pero no son observados por otros procesadores en orden de programa. Como el procesador no espera confirmación de memoria cuando ejecuta una instrucción store, la red de interconexión determina que el orden observado por otros procesadores sea distinto del orden de programa[13].

Confirmación de una escritura. En la Figura 3.12.b se muestra un diagrama temporal donde un procesador, al emitir una instrucción store, espera confirmación de memoria, antes de ejecutar la siguiente instrucción de acceso a memoria[14]. De esta forma garantiza que el orden de programa sea observado, sin modificación, por los otros procesadores.

COHERENCIA DE CACHE

Para reducir la latencia de acceso a memoria se explota la propiedad de localidad de los programas mediante una jerarquía de memoria.

En un multiprocesador interesa también reducir el ancho de banda demandado por los procesadores en la red de interconexión. Por ello se ubican niveles de caches antes de la red de interconexión (caches privadas). Por otro

13. Caminos distintos, longitud distinta, latencia distinta y congestión en los enlaces.
14. Esta confirmación se representa mediante una línea gruesa de trazo discontinuo desde el módulo de memoria al procesador.

lado, esta disposición habilita la posibilidad de que lecturas simultáneas a la misma posición de memoria, por parte de varios procesadores, se efectúen en paralelo.

En la parte izquierda de la Figura 3.13 se muestra un ejemplo donde dos procesadores tiene copia en su cache privada de la variable A. Los dos procesadores pueden leer el valor de la variable A concurrentemente, utilizando la copia en la cache privada. No se utiliza la red de interconexión y no se accede a memoria. La latencia de acceso al dato es la latencia de lectura en la cache privada.

Ahora bien, en un multiprocesador cada procesador observa la memoria global a través de una jerarquía de memoria distinta. En la parte derecha de la Figura 3.13 se muestra que la jerarquía de memoria del procesador P1 es L11-Memoria y la jerarquía del procesador P2 es L12-Memoria. En estas condiciones es factible, aunque no es lo que se desea, que una lectura no devuelva el último valor que se ha escrito en una posición de memoria.

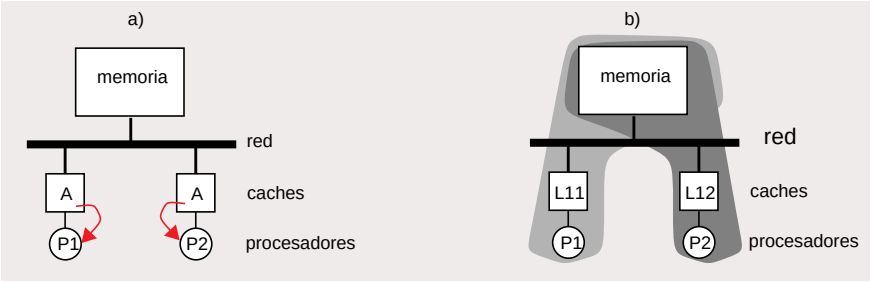


Figura 3.13 Sistema multiprocesador con procesadores que tienen cache privada. a) Lecturas concurrentes a copias de la variable en la cache privada. b) Jerarquía de memoria utilizada por cada procesador para observar la memoria global.

El problema que se plantea no se resuelve con las técnicas utilizadas en la jerarquía de memoria de un sistema uniprocesador (escritura retardada o escritura inmediata). Tampoco se resuelve utilizando los mecanismos empleados en un uniprocesador cuando se efectúa comunicación con el exterior del computador. Esto es, entrada/salida de información del computador. Los mecanismos que se suelen utilizar son groseros y no son eficaces en un sistema multiprocesador por la penalización que introducen.

Ejemplo. Para mostrar la necesidad de la coherencia de cache, utilizaremos como ejemplo la migración de procesos entre procesadores en un multiprocesador con cache privadas. En la Figura 3.14 se muestra un multiprocesador con dos procesadores y un proceso que migra entre los procesadores. Las caches privadas utilizan escritura retardada.

Supongamos que la variable T no está almacenada en ninguna cache inicialmente. Cuando el proceso se ejecuta en el procesador P1 lee la variable T y se produce un fallo de cache. El bloque que contiene la variable se lee de memoria y se almacena en un contenedor de su cache (Figura 3.14.a). Posteriormente el proceso migra al procesador P2 y lee la variable T. Se produce un fallo, el bloque que contiene la variable se lee de memoria y se almacena en un contenedor de su cache (Figura 3.14.b). Ahora hay 2 copias del bloque en las caches y las 2 copias tienen el mismo valor. Seguidamente, mientras el proceso se ejecuta en el procesador P2, se escribe la variable T (Figura 3.14.c). Ahora las 2 copias tienen valores distintos.

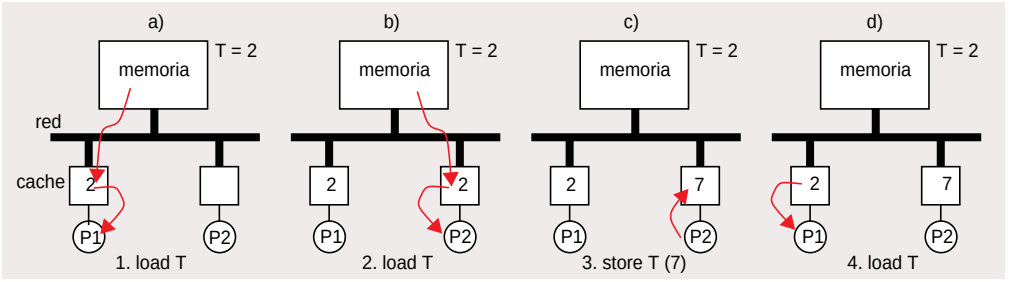


Figura 3.14 Sistema multiprocesador donde un proceso migra entre 2 procesadores.

Después de un intervalo de tiempo, el proceso migra al procesador P1 y lee la variable T. La variable T está almacenada en un contenedor de su cache y por tanto es un acierto (Figura 3.14.d). Ahora bien, el valor de la variable T ya no es el último valor que ha escrito el proceso en la variable T. Este valor está almacenado en la copia residente en la cache del procesador P2. Por tanto, la ejecución será incorrecta, ya que se lee un valor antiguo.

En la Figura 3.15 se muestra en cada fila una fotografía de las caches y memoria para cada acceso a memoria. Las filas están en orden temporal de arriba a abajo, representando en la primera fila el primer acceso.

Actividad del procesador	Actividad de la red	Cache de P1 (valor)	Cache de P2 (valor)	Memoria (valor)	Coherencia en copias
				2	
1. P1 load T	fallo	2	no hay copia	2	coherente
2. P2 load T	fallo	2	2	2	coherente
3. P2 store T		2	7	2	incoherente
4. P1 load T		2	7	2	incoherente

Figura 3.15 Tabla que muestra temporalmente el valor de las copias en un sistema multiprocesador.

En resumen, en un multiprocesador hay que diseñar un mecanismo que mantenga la coherencia entre las jerarquías de memoria utilizadas por cada procesador para observar la memoria global. En concreto, mantener la coherencia de los datos en las caches privadas. Este mecanismo debe ser transparente al lenguaje máquina y debe permitir la replicación y migración de datos entre caches privadas.

Protocolo de coherencia de cache

El objetivo de un protocolo de coherencia de cache hardware es detectar y eliminar las incoherencias. Para ello, las incoherencias deben reconocerse en ejecución.

Propagación de escrituras. Un protocolo de coherencia de cache es simplemente un mecanismo que propaga una operación de escritura a las copias en cache de la posición de memoria actualizada.

Políticas de actuación para propagar una escritura. Distinguiremos dos políticas, que se diferencian en el tipo de actuación al propagar una operación de escritura.

- Actualización (actualización en escritura). En una operación de escritura se actualizan todas las copias en las caches de otros procesadores y en la posición de almacenamiento en memoria.
- Invalidación (invalidación en escritura). En una operación de escritura se invalidan todas las copias en las caches de otros procesadores y en la posición de almacenamiento en memoria.

En la siguiente descripción de las políticas de coherencia, se supone que en cada instante sólo se está efectuando la acción que se describe. La idea no es mostrar una implementación, sino efectuar una descripción a nivel conceptual de la política.

En los dos casos que se muestran supondremos un multiprocesador donde los procesadores utilizan escritura retardada en la jerarquía de memoria observada. Como ejemplo se utilizará el mismo que en la Figura 3.14, donde un proceso migra entre procesadores.

Propagación de escrituras: política de actualización

En una política de actualización la granularidad de la acción de coherencia es la palabra. En la Figura 3.16.a y la Figura 3.16.b el proceso efectúa la lectura de la variable T y produce un fallo de cache en los procesadores P1 y P2. Cuando el procesador P2 efectúa la operación de escritura se transmite por la red esta intención y el valor que debe utilizarse para actualizar la variable T. La

cache del procesador P1 tiene una copia de la variable T y al observar, o conocer de alguna forma, esta acción actualiza su copia, utilizando el valor transmitido por la red (la escritura se propaga). Suponemos que memoria también se actualiza.

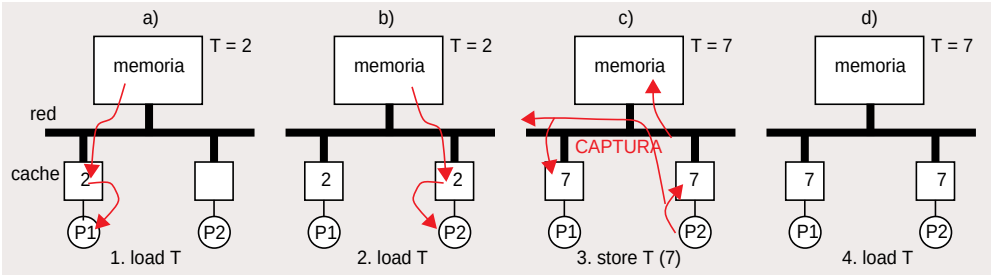


Figura 3.16 Política de actualización para mantener la coherencia.

Lecturas posteriores de la variable T, por parte de cualquiera de los dos procesadores, son acierto en cache. Una escritura de la variable T por parte del procesador P1 o P2, desencadena nuevamente una transmisión de la intención por la red, junto con el valor con el cual se actualiza la variable T. La otra cache, al conocer la intención actualiza la variable T.

Propagación de escrituras: política de invalidación

En una política de invalidación la granularidad de la acción de coherencia es el bloque. En la Figura 3.17.a y la Figura 3.17.b el proceso efectúa la lectura de la variable T y produce un fallo de cache en los procesadores P1 y P2. Cuando el procesador P2 efectúa la operación de escritura se transmite por la red esta intención, la cual es observada, o conocida de alguna forma, por la otra cache y memoria. Esta intención desencadena la invalidación de la copia del bloque, que contiene la variable T, en la cache de otros procesadores e implícitamente en memoria (Figura 3.17.c). Esto es, la escritura se propaga. Posteriormente se actualiza en la cache de P2.

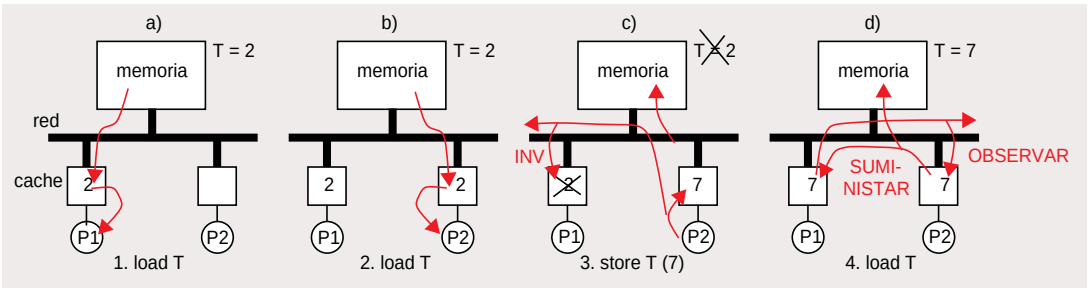


Figura 3.17 Política de invalidación para mantener la coherencia.

Posibles operaciones posteriores, de lectura o escritura, del procesador P2 no desencadenan ninguna otra acción, ya que en la cache de P2 se dispone de información, para determinar que es la única copia válida del bloque y se dispone de derechos de acceso para leer y escribir.

La política de invalidación requiere de un mecanismo que suministre la copia válida del bloque en una operación de lectura. Cuando el procesador P1 quiere leer la variable T, se detecta un fallo de cache y se transmite por la red la intención de lectura del bloque que contiene la variable T. Esta intención es observada, o conocida de alguna forma, por la cache del procesador P2, la cual suministra una copia del bloque de memoria a la cache del procesador P1. Adicionalmente, se actualiza el bloque en memoria. Lecturas posteriores de la variable T, por parte de los procesadores P1 y P2, son aciertos en sus respectivas cache. Una escritura de la variable T, por parte del procesador P1 o P2, desencadena nuevamente una invalidación de la otra copia del bloque que contiene la variable.

Notemos que para propagar un valor, la política de invalidación induce un fallo en las caches que tienen copia del bloque, que contiene la variable que se escribe. Entonces, al servirse el fallo se suministra una copia del bloque con el valor actualizado.

Tipos de protocolos de coherencia de cache

Los protocolos de coherencia se sustentan en seguir la pista del estado de cualquier bloque de memoria[15]. Se utilizan básicamente dos clases de protocolos de coherencia, los cuales utilizan técnicas distintas para seguir la pista del estado de un bloque de memoria.

En la Figura 3.18 se muestra un esquema de un sistema basado en directorio y de un esquema basado en observación o difusión.

Basado en directorio. El estado de un bloque de memoria se mantiene en una sola ubicación, denominada directorio. El directorio tiene información de la ubicación de las copias y el estado[16]. Las peticiones de los controladores de cache o de coherencia se encaminan mediante una red de interconexión al directorio. El directorio, a partir de la información de estado e identificación de posibles réplicas, toma las acciones oportunas para servir la solicitud[17]. Sólo

15. La granularidad del estado es a nivel de bloque, aunque un acceso a memoria tiene menor granularidad. Entonces, un acceso a memoria puede modificar el estado del bloque.
16. Por estado entendemos, en general, si hay copias y las operaciones que pueden realizarse en cada una de estas copias.
17. Peticiones a los controladores de coherencia que éstos responden.

se establece comunicación con los controladores de cache que tienen copia del bloque de memoria (Figura 3.18.a). El directorio, después de recibir las respuestas de los controladores de cache, responde al solicitante.

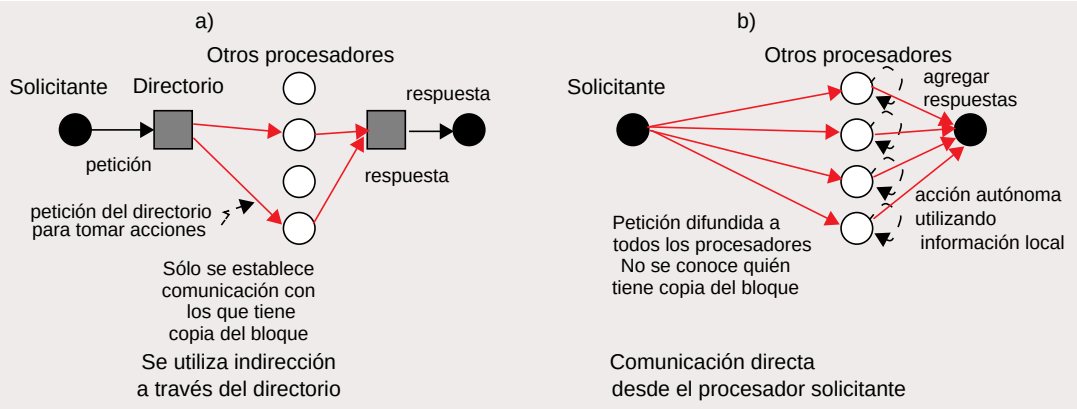


Figura 3.18 Tipos de protocolo de coherencia de cache: a) basado en directorio y b) basado en difusión u observación.

Basado en observación. Cada cache, que tiene una copia de un bloque de memoria, también tiene una copia del estado de compartición del bloque. No se mantiene ningún estado centralizado. Por tanto, la información de estado está distribuida. Las caches son accesibles mediante una red de interconexión, que permite difundir información (petición, respuesta). Todos los controladores de cache monitorizan u observan la información transmitida por la red de interconexión para determinar, a partir de la información difundida, si ellos tienen o no copia del bloque que es accedido. El resultado de la observación se agrega (agrupa) y se suministra al solicitante para que determine el estado de su copia del bloque. Además, de forma autónoma y utilizando información local, cada controlador de cache toma la acción oportuna, para actualizar el estado y servir la petición difundida, si es el caso (Figura 3.18.b). Memoria suministra el bloque cuando no es suministrado por una cache.

Consideraciones de implementación

La jerarquía de memoria es transparente al programador. Entonces, el comportamiento que observan los hilos, al acceder a una posición de memoria, debe ser el mismo que cuando no existen caches privadas. Una alternativa es permitir sólo una copia. Otra alternativa es propagar una escritura instantáneamente a todas las copias. Esto es, una escritura se propaga de forma atómica.

La propagación de una escritura de forma atómica establece un orden global serie de las escrituras a una posición de memoria[18]. En una lectura de una posición de memoria se devuelve el valor de la última escritura serializada, cuando está consolidada. Esto es, el valor de la escritura es observable por todos los procesadores[19].

En la Figura 3.19 se muestra un ejemplo con cuatro hilos. Todos los accesos son a la misma posición de memoria. Las instrucciones load en los hilos H3 y H4 deben observar las escrituras en el mismo orden. Por ejemplo, si el primer load del hilo H3 lee el valor escrito por el hilo H2 (8) y el segundo load, del mismo hilo, lee el valor escrito por el hilo H1 (3), no es posible que el primer load del hilo H4 lea el valor 3 y el segundo load, del mismo hilo, lea el valor 8.

Hilo H1	Hilo H2	Hilo H3	Hilo H4
store R4, A	store R2, A	load R6, A	load R9, A
		load R7, A	load R10, A

Figura 3.19 Propagación de escrituras. Cuatro hilos que acceden a la misma posición de memoria. El valor de los registros R4 y R2 es 3 y 8 respectivamente.

La propagación de una escritura no puede considerarse una acción instantánea o atómica, ya que en un multiprocesador la transmisión de información no es en tiempo cero[20].

Una forma de implementar una operación atómica es no permitir que los cambios que se producen progresivamente (subacciones) sean observados individualmente[21]. Todos los cambios serán visibles por todos los procesadores utilizando una única acción, cuando la operación ha finalizado (consolidado).

Son necesarios dos mecanismos para implementar la atomicidad de una escritura a una posición de memoria: a) identificar un punto o lugar de serialización[22] y b) una técnica para determinar la consolidación de una escritura (la propagación ha finalizado).

18. De forma idéntica a un multiprocesador sin cache, donde las escrituras consolidan en el módulo de memoria.
19. La atomicidad en cada escritura es una condición suficiente desde el punto de vista de coherencia de cache. La propagación de una escritura de forma atómica establece un orden global serie de las escrituras a una posición de memoria. Ahora bien, es posible relajar la condición y seguir manteniendo coherencia de cache. Esta relajación es útil en multiprocesadores con consistencia relajada de memoria.
20. Las lecturas cumplen la propiedad de idempotencia. El resultado de varias lecturas a la misma posición de memoria es el mismo que el de una lectura.
21. Por ejemplo, el valor que establece una escritura puede ser observado por alguno o algunos procesadores mientras que otros están observan un valor distinto.
22. Este punto o lugar de ordenación puede ser el mismo para todas las posiciones de memoria o un lugar distinto para cada posición de memoria.

Punto, entidad o lugar de serialización. Dada una posición de memoria, ésta tiene un lugar que se utiliza para serializar las escrituras[23]. Todos los procesadores encaminan los accesos a memoria a este punto de serialización. Esto es, esta entidad observa los accesos de todos los procesadores a la posición de memoria. Desde este punto de serialización se inician las acciones necesarias para mantener la coherencia.

Consolidación o finalización de la propagación de una escritura. Todas las caches deben responden a la solicitud de coherencia efectuada desde el lugar de serialización. Una vez se han recibido todas las respuestas puede considerarse que la escritura ha consolidado[24] [25].

La consolidación de una escritura se utiliza para[26]:

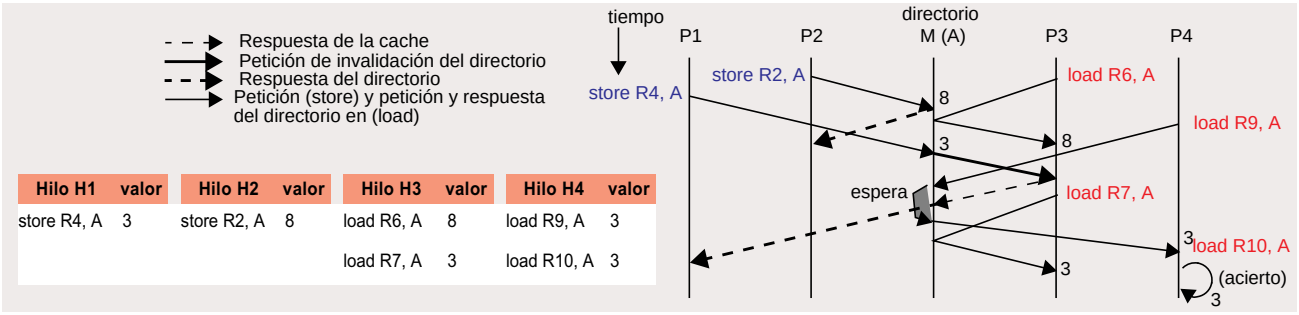
- Conocer cuándo el valor de una escritura es observable por todos los procesadores[27]. A partir de este instante se puede suministrar el valor a una instrucción load.
- Cuándo puede iniciarse el procesado de otra escritura.

Seguidamente se muestran dos ejemplos. Se utiliza un multiprocesador con caches privadas, que utilizan escritura inmediata sin asignación de contenedor en un fallo de escritura.

Para mantener la coherencia se utiliza un directorio y la política es de invalidación. El directorio confirma la escritura a la cache solicitante cuando la escritura ha sido consolidada. Para ello, el directorio espera las respuestas a las peticiones de invalidación, que ha efectuado a las caches que tienen copia. Suponemos que el directorio envía la señal de consolidación o confirmación, aunque conozca que no hay copia del bloque en el solicitante. También, suponemos que siempre se utiliza el mismo camino entre un emisor y un receptor y que en el camino se mantiene el orden de los mensajes.

23. En un multiprocesador sin cache privadas el punto de serialización es la memoria. En un multiprocesador con cache y un protocolo de directorio se puede utilizar el directorio como punto de serialización.
24. En función de propiedades de la red de interconexión no es necesario una respuesta explícita, está implícita en alguna acción efectuada en el punto de ordenación. Por otro lado, en una red que no mantiene el orden de mensajes entre emisor y destino pueden ser necesarios más mensajes o gestionar el desorden de la red en la implementación del protocolo de coherencia.
25. Un controlador de cache en una operación de lectura, mientras no recibe una acción de coherencia, suministra el valor almacenado en cache.
26. Desde el punto de vista de consistencia secuencial se utiliza para implementar orden de programa.
27. En un protocolo de coherencia con política de invalidación, el valor de una escritura no puede ser observado por un load hasta que han sido confirmadas todas las invalidaciones. En un protocolo de coherencia con política de actualización son necesarios dos pasos. En el primero se reciben las confirmaciones y posteriormente se indica a los nodos, involucrados en la acción de coherencia, que el nuevo valor es observable. En el intervalo temporal hay que esperar.

Ejemplo. En la Figura 3.20 se muestra una ejecución de los hilos de la Figura 3.19. Las caches no tiene copia de la variable A. Al procesar en el directorio la instrucción store de P1 hay que invalidar la copia en la cache de P3. El directorio espera, a que se consolide la escritura en curso, para responder a la primera instrucción load del procesador P4. La segunda instrucción load en el procesador P3 es un fallo de cache. La segunda instrucción load en el procesador P4 es un acierto en cache.

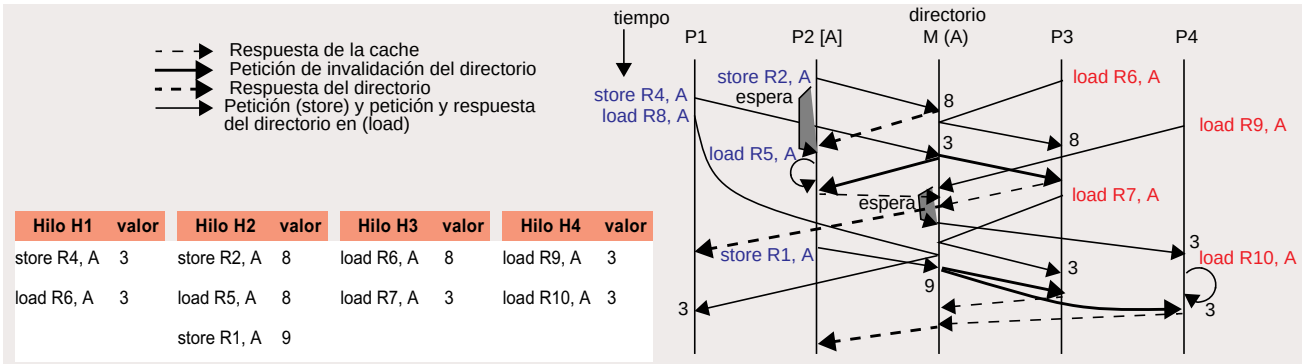


Hilo H1	valor	Hilo H2	valor	Hilo H3	valor	Hilo H4	valor
store R4, A	3	store R2, A	8	load R6, A	8	load R9, A	3
				load R7, A	3	load R10, A	3

Figura 3.20 Coherencia de cache. Propagación de escrituras I.

Ejemplo. En la Figura 3.21 se parte del código de la Figura 3.19 y los hilos H1 y H2 ejecutan instrucciones load después de las instrucciones store. Adicionalmente el hilo H2 ejecuta una instrucción store antes de finalizar. Por otro lado, el procesador P2 tiene copia de la variable A en la cache.

En el procesador P2 se espera la confirmación de que el store ha consolidado antes de ejecutar la instrucción load, la cual es un acierto en cache. La cache se actualiza al recibir la confirmación (la escritura es globalmente visible). La instrucción load del procesador P1 es un fallo en cache. El procesador efectúa la petición del bloque antes de recibir la confirmación de la instrucción store[28].



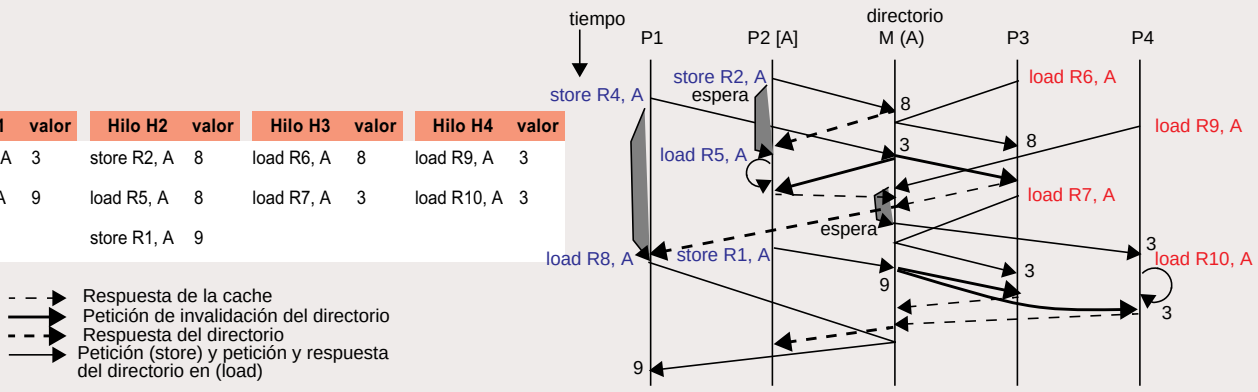
Hilo H1	valor	Hilo H2	valor	Hilo H3	valor	Hilo H4	valor
store R4, A	3	store R2, A	8	load R6, A	8	load R9, A	3
load R6, A	3	load R5, A	8	load R7, A	3	load R10, A	3
		store R1, A	9				

Figura 3.21 Coherencia de cache. Propagación de escrituras II.

Ejercicio *En el multiprocesador descrito previamente suponga que el camino utilizado entre un emisor y un receptor puede ser distinto para cada petición. Muestre un diagrama temporal de la ejecución del código de la Figura 3.21.*

Respuesta En la siguiente figura se muestra el diagrama temporal de ejecución.

Hilo H1	valor	Hilo H2	valor	Hilo H3	valor	Hilo H4	valor
store R4, A	3	store R2, A	8	load R6, A	8	load R9, A	3
load R6, A	9	load R5, A	8	load R7, A	3	load R10, A	3
		store R1, A	9				



Como la red puede utilizar caminos distintos, el acceso debido a la instrucción load del hilo H1 podría llegar al directorio antes que el acceso debido a la instrucción store. Para respetar las dependencias de datos en el hilo H1 es necesario esperar la confirmación de la escritura.

MULTIPROCESADOR SECUENCIALMENTE CONSISTENTE CON CACHE PRIVADAS

En el apartado de consistencia de memoria se ha mostrado que, para establecer un orden de accesos a una posición de memoria, efectuados por procesadores distintos, el sistema de memoria debe respetar el orden de lecturas y escrituras a diferentes posiciones de memoria, efectuadas desde un hilo (por ejemplo en la Figura 3.10). Coherencia de cache no establece relaciones entre accesos a posiciones de memoria distintas. Entonces, las condiciones de coherencia de cache son insuficientes para disponer de consistencia secuencial

Seguidamente se muestran las condiciones de consistencia secuencial en un multiprocesador con cache privadas, detallando los requisitos.

28. Notemos que no se espera a que la instrucción store esté consolidada. Ahora bien, hay que respetar el orden de accesos a la misma posición de memoria. Son dependencias de datos del hilo. Como se utiliza el mismo camino entre emisor y receptor y se mantiene el orden de los mensajes, está garantizado que se respetan las dependencias del hilo. La instrucción load se procesa en el directorio después de la instrucción store del mismo procesador. En general, entre ellas dos, en el directorio, pueden procesarse otros accesos a la misma posición de memoria.

Orden de programa. Un procesador debe esperar a que el anterior acceso a memoria se haya consolidado antes de iniciar el siguiente acceso a memoria en orden de programa.

- Finalización de una lectura. Cuando el dato llega al procesador.
- Finalización de una escritura. Conocer que la escritura ha sido consolidada. Esto es, han sido confirmadas todas las acciones de propagación de la escritura por las caches destinatarias[29]. Por tanto, la escritura ha sido serializada.

Es un mecanismo local del procesador que requiere conocer las confirmaciones de las acciones de propagación de una escritura.

Atomicidad. Todos los procesadores observan los accesos a memoria en el mismo orden.

- Atomicidad de una escritura. Conocer que una escritura ha sido consolidada, para procesar la siguiente escritura si es el caso. Por otro lado, una lectura obtiene el valor de la última escritura consolidada[30]. Esto es, el valor de una escritura sólo es observable cuando todas las acciones de coherencia han sido confirmadas. Por tanto, todos los procesadores observan las escrituras a la misma posición de memoria en el mismo orden[31].

Es un mecanismo disponible para gestionar cada bloque (unidad de gestión).

Orden de programa

Seguidamente se muestra, con un ejemplo, que el inicio del procesado en el directorio de una escritura no es una condición suficiente para garantizar que el orden de programa en un procesador es observado por los otros procesadores.

En la Figura 3.22 se muestra la organización del multiprocesador. La red y el encaminamiento de mensajes es el mismo que el descrito para el multiprocesador de la Figura 3.11. Los nodos disponen de un procesador, de una cache privada y de un módulo de memoria. La cache privada utiliza escritura inmediata. Los módulos de memoria se entrelazan de la misma forma que en la Figura 3.11.

29. Por ejemplo, el directorio puede ser el encargado de recolectar las respuestas de las caches a las peticiones de invalidación del directorio. Una vez han respondido todas las caches, el directorio confirma la escritura al procesador solicitante.
30. Si una escritura está pendiente de consolidación, una alternativa es esperar la consolidación para suministrar el valor a la lectura.
31. En el apartado “Necesidades del modelo de consistencia secuencial” se concluye que es suficiente con que las escrituras a cada posición de memoria sean atómicas.

Suponemos un protocolo de coherencia de tipo directorio y con política de invalidación. Un acceso a memoria, que no puede servirse localmente, accede al directorio[32], el cual está ubicado en el mismo nodo que el módulo de memoria donde se ubica la variable.

El código que se ejecuta en el multiprocesador es una sincronización por evento (Figura 3.10). La variable aviso está ubicada en la memoria del nodo cero y la variable A en la memoria del nodo 8. La cache del nodo 2 almacena las dos variables.

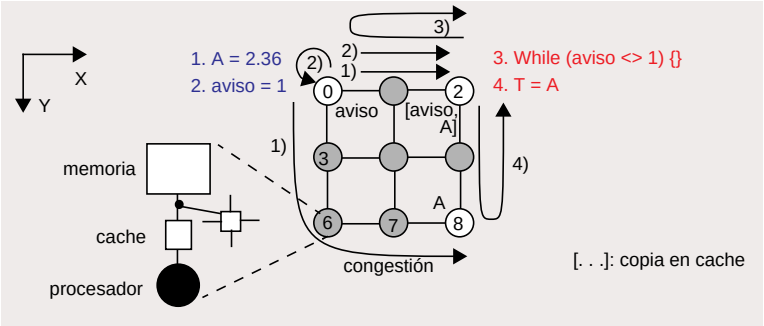


Figura 3.22 Multiprocesador cuyos nodos tienen cache privada. Orden de programa. Necesidad de esperar las respuestas a una acción de coherencia (no se muestran).

Para simplificar el diagrama temporal no se muestran los accesos debidos a la espera activa en el procesador P2.

En la parte izquierda de la Figura 3.23 se muestra el caso en el cual el directorio responde al emitir la acción de coherencia a la cache involucrada. Esto es, sin esperar la respuesta de la cache involucrada en la acción de coherencia. El procesador P0 utiliza esta respuesta para ejecutar la siguiente escritura (aviso). El resultado en el diagrama temporal de la Figura 3.23.a no es secuencialmente consistente. El valor leído de A en el procesador P2 es un valor anterior al escrito por el procesador cero[33]. La escritura de la variable aviso es observada en el procesador P2 antes que la escritura de la variable A. Las dos escrituras son efectuadas por el procesador P0, pero el orden de programa observado ha sido intercambiado por la red de interconexión.

En la parte derecha de la misma Figura 3.23 se muestra el caso en el cual el directorio espera la respuesta, de la cache involucrada, antes de emitir la respuesta a la cache que ha efectuado la petición. Esto es, la propagación ha consolidado y la escritura ha consolidado. Cuando el procesador P0 ejecuta la siguiente escritura, la escritura previa es globalmente visible. El resultado en el

32. Punto o entidad de serialización.
33. Notemos que las caches se mantienen coherentes. En algún instante la copia en la cache del nodo 2 es invalidada.

diagrama temporal de la Figura 3.23.b es secuencialmente consistente. El procesador P2 observa el orden de programa del hilo que se ejecuta en el procesador P0[34].

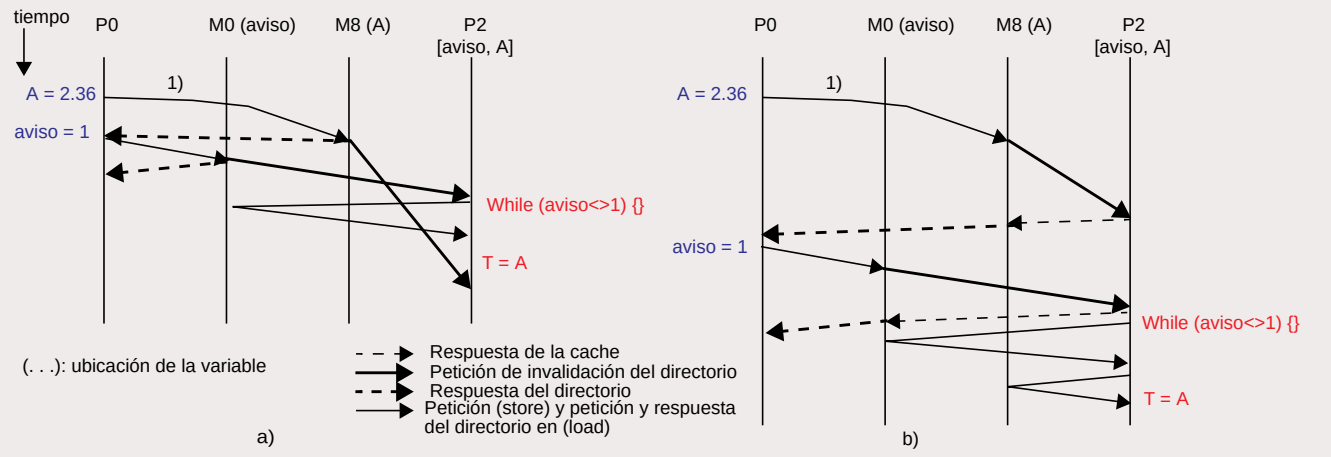


Figura 3.23 Necesidad de esperar las respuestas de las caches a una acción de coherencia: a) no se espera la respuesta de las caches, b) se espera la respuesta de las caches.

Atomicidad de la escritura

La atomicidad de los accesos a una posición de memoria, inducido por el modelo de consistencia secuencial, determina que el valor leído en una instrucción load es el valor escrito por la última instrucción store consolidada.

Suministro del valor de una escritura. El valor que establece una escritura (visibilidad) sólo puede suministrarse después de que todas las acciones de coherencia han sido confirmadas. Esto es, la propagación de la escritura ha finalizado[35].

Para mostrar la necesidad de atomicidad en las escrituras utilizaremos como ejemplo el código de la Figura 3.24. El multiprocesador que se utiliza, la red y el encaminamiento de mensajes es el mismo que el descrito para el multiprocesador de la Figura 3.11 (Figura 3.24). Los nodos disponen de un procesador, de una cache privada y de un módulo de memoria. La cache privada utiliza escritura inmediata. Los módulos de memoria se entrelazan de la misma forma que en la Figura 3.11.

34. Otra alternativa es que el directorio no responda a la lectura, de la variable aviso del procesador P2, hasta que recibe la respuesta de invalidación del bloque que contiene la variable A (Figura 3.23.a). Esta alternativa reduce la serialización en el procesador P0.
35. En un protocolo de coherencia con política de invalidación, el valor de una escritura no puede ser observado por un load hasta que han sido confirmadas todas las invalidaciones.

Suponemos un protocolo de coherencia de tipo directorio y con política de invalidación. Un acceso a memoria, que no puede servirse localmente, accede al directorio, el cual está ubicado en el mismo nodo que el módulo de memoria donde se ubica la variable.

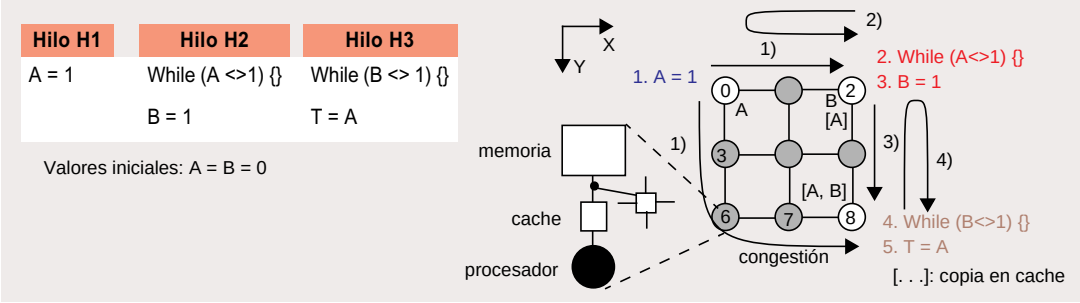


Figura 3.24 Requisito de atomicidad de las escrituras. Secuencias de código y organización del multiprocesador. Las respuestas a una acción de coherencia no se muestran.

El código consta de tres hilos (parte izquierda de la Figura 3.24). El hilo H1 se ejecuta en el procesador del nodo cero y los hilos H2 y H3 en los procesadores de los nodos 2 y 8 respectivamente.

La variables A y B están ubicadas en la memoria de los nodos 0 y 2 respectivamente. En las cache de los nodos 2 y 8 hay copia de la variable A y en la cache del nodo 8 hay copia de la variable B.

En el camino que comunica el nodo cero con el nodo 8 hay congestión. Esto es, los mensajes se retrasan por la comunicación existente entre otros nodos.

En la Figura 3.25 se muestran los casos relativos al suministro de un dato en una instrucción load: a) no se espera la confirmación de la propagación de la escritura a todas las cache involucradas en la acción de coherencia previa (la escritura aún no ha consolidado, Figura 3.25.a) y b) se espera la confirmación de la propagación de la escritura a todas las cache involucradas en las acción de coherencia previa (la escritura ha consolidado, Figura 3.25.b).

Para reducir el tamaño de los diagramas temporales no se muestran con la misma escala de tiempo (mensajes de invalidación desde el nodo 0). Tampoco se muestran los accesos debidos a la espera activa en los procesador P2 y P8[36].

36. Notemos que hasta que la escritura de un procesador no es propagada a todos los otros procesadores (ha consolidado), una lectura de estos últimos lee el valor previo que hay almacenado en cache. Una vez consolidada la propagación de la escritura todos los procesadores leen el mismo valor.

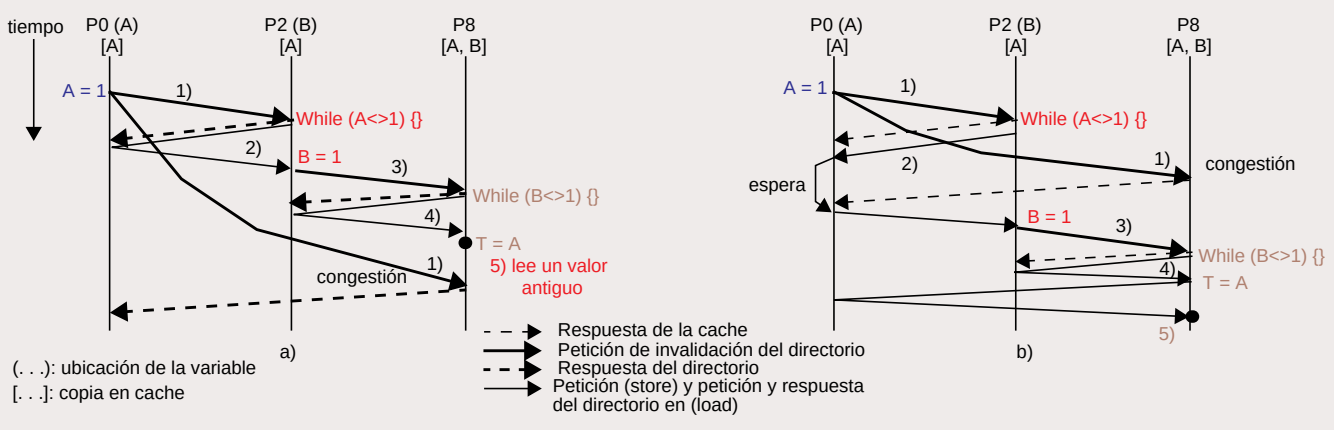


Figura 3.25 Diagrama temporales. Requisito de atomicidad de las escrituras.

En el nodo cero, al ejecutarse la instrucción store, el directorio emite mensajes de invalidación, del bloque que contiene la variable A, a los nodos 2 y 8. En el nodo 2, después de recibir y procesar la invalidación se produce un fallo de cache al leer la variable A. El controlador de coherencia emite un mensaje de petición de bloque al nodo cero, en cuyo módulo de memoria está ubicada la variable A. En el caso a) la memoria del nodo cero responde inmediatamente a la petición del nodo 2 (Figura 3.25.a). En cambio, en el caso b) la respuesta a la petición del nodo 2 se efectúa después de que se hayan recibido todas las confirmaciones a las peticiones de invalidación (nodos 2 y 8, Figura 3.25.b).

En el diagrama temporal de la Figura 3.25.a el valor leído de A en el procesador P8 es un valor previo al escrito por el procesador P0. No existe atomicidad en las escrituras, ya que el procesador P2 ha leído el valor uno[37]. En el diagrama temporal de la Figura 3.25.b se garantiza que la escritura de A por el procesador P0 se observa de forma atómica. No se suministra el valor hasta que se han recibido todas las confirmaciones a las invalidaciones emitidas.

CONSISTENCIA DE MEMORIA RELAJADA

La semántica de consistencia secuencial es intuitiva[38], pero es demasiado estricta: a) mantener orden de programa y b) atomicidad de las escrituras. Todo ello limita: a) implementaciones hardware y b) optimizaciones del compilador, cuyo objetivo es mejorar el rendimiento. Por ello, la mayoría de procesadores implementan un modelo más relajado de consistencia de memoria[39].

37. Observemos que la cache de P8 será coherente una vez llegue la invalidación.

En la Figura 3.26 se muestra un ejemplo donde el modelo de consistencia secuencial es demasiado estricto desde el punto de vista del resultado que espera obtener el programador (semántica). El análisis lo efectuamos desde el punto de vista de las posibilidades de reordenación de instrucciones de un compilador.

En la parte izquierda de la Figura 3.26 se muestran, mediante líneas finalizadas con flecha, las restricciones de orden del modelo de consistencia secuencial. En la parte derecha de la misma figura se muestran las restricciones de orden que garantizan la semántica que espera el programador. En este caso, las escrituras y las lecturas de las variables A y B pueden reordenarse entre ellas. Las únicas restricciones son que: a) las escrituras se especifiquen antes que la escritura a la variable aviso en H1 y b) las lecturas se especifiquen después de la lectura de la variable aviso en H2.

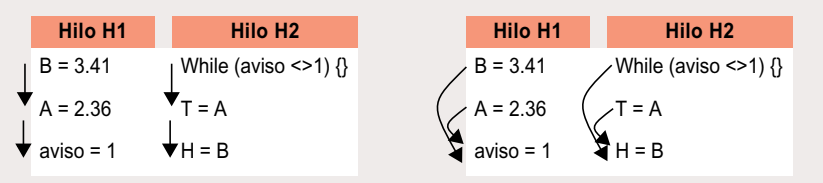


Figura 3.26 Código para mostrar que la semántica de modelo de consistencia secuencial es demasiado estricta. El valor inicial de la variable aviso es cero.

Lenguajes y compilación

Supondremos que el lenguaje de alto nivel utilizado para especificar los programas paralelos implementa un modelo relajado de consistencia[40].

Para establecer consistencia secuencial entre regiones de código utilizaremos la directiva LNbarrera[41]. Esta directiva define un punto, en la secuencia de instrucciones, en el cual se garantiza que un hilo consolida sus accesos a

38. Se adecua al modelo de un sistema uniprocesador multiprogramado.
39. En procesadores donde se utiliza especulación al ejecutar instrucciones, las implementaciones hardware, cuyo objetivo es mejorar el rendimiento del procesador, pueden hacerse transparentes al programador. Esto es, el modelo de memoria es secuencialmente consistente.
40. No se respeta ninguno de los ordenes de programa (Figura 3.7).
41. Lenguajes como OpenMP implementan un modelo relajado de consistencia. Para establecer consistencia secuencial entre regiones de código se utiliza el pragma o directiva FLUSH. !$OMP FLUSH (list) en Fortran y #pragma omp flush (list) en C/C++. Esta directiva está implícita en otras directivas. Un hilo pueden mantener una visión de la memoria que temporalmente no es secuencialmente consistente con la que observan otros hilos. Estas visiones temporales se pueden hacer secuencialmente consistentes en ciertos puntos del programa. La operación que fuerza que los accesos a memoria de un hilo, hasta ese punto, sean observados por otros hilos se denomina flush.

memoria: a) todos los accesos previos a memoria han consolidado y son observables por los otros hilos, b) no se han ejecutado accesos a memoria especificados posteriormente en el hilo.

La directiva LNbarrera además de limitar las optimizaciones del compilador genera una instrucción de lenguaje máquina, denominada INbarrera[42] (fence, sync, etc). Esta instrucción es utilizada por la arquitectura, si el modelo de consistencia que soporta no es secuencialmente consistente.

Nosotros no analizaremos en detalle la semántica de ningún lenguaje. Unicamente nos centraremos en los puntos donde interesa que se disponga de consistencia secuencial. Por ello, utilizaremos la primitiva LNbarrera para indicar que el compilador: a) no puede reordenar instrucciones o asignar una variable a un registro y b) genere una instrucción INbarrera.

Para garantizar que la escritura de una variable en un hilo sea consolidada y otro hilo observe este valor, deben de efectuarse las siguientes operaciones: a) el hilo H1 escribe la variable V, b) el hilo H1 ejecuta la instrucción generada al incluir la directiva LNbarrera (instrucción INbarrera), c) el hilo H2 ejecuta la instrucción generada al incluir la directiva LNbarrera (instrucción INbarrera) y d) el hilo H2 lee la variable V.

Seguidamente utilizamos un ejemplo para mostrar la utilización de la directiva LNbarrera. En el código de la parte izquierda de la Figura 3.27, por ejemplo, el compilador puede reordenar las lecturas y escrituras a cualquier variable. También puede asignar la variable aviso a un registro.

Hilo H1	Hilo H2
a. A = 2.36	c. While (aviso <> 1) {}
b. aviso = 1	d. T = A
Valor inicial : aviso = 0	

Hilo H1	Hilo H2
a. A = 2.36	LNbarrera
LNbarrera	c. While (aviso <> 1) { }
b. aviso = 1	LNbarrera
LNbarrera	d. T = A

Figura 3.27 Secuencias de código para mostrar la utilización de la directiva LNbarrera.

En el código de la parte derecha de la Figura 3.27 se muestra la inserción de la directiva LNbarrera. En el hilo H1 la primera LNbarrera asegura que la escritura de la variable A consolida antes de efectuar la escritura de la variable aviso. La segunda LNbarrera asegura que la escritura de la variable aviso consolida antes de ejecutar alguna instrucción más joven[43]. En el hilo H2, la

42. No hay que confundir la directiva LNbarrera y la instrucción INbarrera con la operación de sincronización que se denomina BARRERA.

43. La inserción en este punto de la primitiva LNbarrera y la generación de la instrucción correspondiente es una acción conservadora. Depende del código que se especifica posteriormente.

primera LNbarrera asegura que la variable aviso se lee de memoria[44]. La segunda LNbarrera asegura que la variable A se lee de memoria, después de que consolide la lectura de la variable aviso.

En función del modelo de consistencia de memoria de la arquitectura, puede no ser necesario la generación de la instrucción INbarrera cuando se especifica la primitiva LNbarrera. Notemos que una arquitectura puede garantizar algunos de los ordenes de programa (Figura 3.7). En el siguiente apartado se analiza un caso.

Optimización hardware

El modelo de consistencia secuencial determina que la latencia de una escritura puede ser significativa. Hay que esperar la confirmación de las acciones de coherencia. Ello representa una pérdida de rendimiento, ya que un procesador no puede ejecutar el siguiente acceso a memoria hasta que ha sido consolidado la escritura.

La relajación del modelo de consistencia, en el cual un acceso de lectura a memoria no debe esperar la consolidación de la escritura previa, en orden de programa, se denomina Total Store Order (TSO) y permite la introducción de un buffer de escrituras (BE) en la microarquitectura del procesador[45] (Figura 3.28).

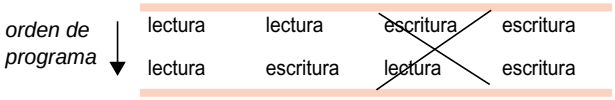


Figura 3.28 Modelo relajado de consistencia de memoria. Total Store Order,

Buffer de escrituras

Los procesadores ejecutan las instrucciones en orden de programa. Para que las escrituras no bloqueen al procesador se utiliza un buffer denominado de escrituras (BE). Una vez una instrucción store ha finalizado la ejecución en el procesador, en orden de ejecución[46], se almacena en el BE. Posteriormente se actualiza memoria.

44. Tengamos en cuenta que el mismo hilo podría haber escrito previamente la variable y estar almacenada en el buffer de escrituras. Se consolida la escritura.
45. En estas condiciones el buffer de escritura no es transparente al programador desde el punto de vista del modelo de consistencia. Si es transparente desde el punto de vista de las dependencias entre instrucciones de acceso a memoria ejecutadas en el procesador. Esto es, en la microarquitectura de un procesador hay hardware que gestiona las dependencia de datos entre las instrucciones de acceso a memoria que ejecuta el procesador.
46. Orden en el cual el procesador ejecuta las instrucciones

Cada entrada del BE tiene un campo de dirección y un campo de dato y el orden en que se almacenan sucesivas instrucciones store en el BE mantiene el orden de programa (parte derecha del la Figura 3.29).

El objetivo de un BE es soportar la latencia de las escrituras. El procesador sigue ejecutando instrucciones. Una instrucción de cálculo actualiza el estado del procesador. Una instrucción store se almacena en el BE. Si la instrucción es un load, se inicia el acceso a memoria adelantando a las instrucciones store almacenadas en el BE. La actualización de memoria desde el BE se efectúa en orden de programa de las instrucciones store.

Para que el BE sea transparente a la arquitectura del procesador[47] se añade circuitería a su alrededor. Si una instrucción load más joven accede a una posición de memoria almacenada en el BE, el procesador obtiene el valor de la instrucción store más joven, que accede a la misma posición de memoria almacenada en el BE[48]. Si el BE está lleno y se inicia la interpretación de una nueva instrucción store, se bloquea la interpretación de instrucciones hasta que se ha vaciado el BE. En la parte izquierda de la Figura 3.29 se muestra un esquema de parte del camino de datos relativo al BE.

Multiprocesador con procesadores que utilizan un BE

En la parte derecha de la Figura 3.29 se muestra un esquema de un multiprocesador con dos procesadores. El multiprocesador que utilizaremos como ejemplo tiene un bus como red de interconexión. El bus está ocupado durante la latencia de acceso a memoria. En cada procesador pueden competir por el bus la entrada en la cabeza del BE y una instrucción load. En un procesador una instrucción load obtiene de forma prioritaria el acceso al bus.

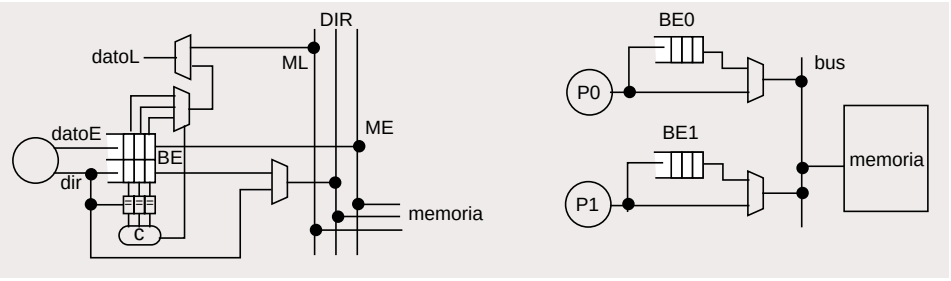


Figura 3.29 Esquema del camino de datos del BE. Multiprocesador cuyos procesadores utilizan un buffer de escrituras (BE).

47. Respeten las dependencia de datos entre instrucciones de acceso a memoria que ejecuta el procesador.
48. Otra alternativa es que el procesador se bloquee mientras perdura el riesgo de datos. El riesgo desaparece cuando la instrucción store, almacenada en el BE, actualiza memoria y se extrae del BE.

La actualización de memoria desde el BE se efectúa en orden de programa de las instrucciones store. Una escritura emitida desde el BE no consolida hasta que recibe la respuesta de memoria[49]. Entonces, se extrae del BE y la siguiente entrada puede competir por acceder al bus[50].

Consistencia secuencial de memoria

Mientras no se emita al sistema de memoria una escritura, a una posición de memoria, ésta no es serializada y en consecuencia no está consolidada. Por tanto, no es observable por otros procesadores. En estas condiciones, la utilización del valor de una escritura para suministrar un valor a una instrucción load, que accede a la misma posición de memoria, determina que la escritura no sea atómica. Por tanto no se cumple consistencia secuencial.

Seguidamente utilizamos un ejemplo para mostrar la utilización de la instrucción INbarrera con el objetivo de establecer consistencia secuencial en puntos del código donde sea necesario.

En el multiprocesador de la Figura 3.29 se ejecuta el esquema de código que se muestra en la parte izquierda de la Figura 3.30, que es el mismo que el mostrado en la Figura 3.8 con la inclusión de la directiva LNBARRERA. En la parte derecha se muestra el código en lenguaje ensamblador.

Hilo H1	Hilo H2
aviso1 = 1	aviso2 = 1
LNbarrera	LNbarrera
T = aviso2	H = aviso1

Valores iniciales: aviso1 = aviso2 = 0

Hilo H1	Hilo H2
store R4, aviso1	store R5, aviso2
INbarrera	INbarrera
load R6, aviso2	load R7, aviso1

Figura 3.30 Esquemas de código para mostrar la gestión del BE cuando se interpreta una instrucción barrera. Los registros R4 y R5 almacenan el valor uno.

En la parte izquierda de la Figura 3.31 se muestra un diagrama temporal de los accesos a memoria al ejecutar el código de la Figura 3.30 sin las instrucciones INbarrera. El primer acceso de cada procesador es una escritura. Cada una de ellas se almacena en el BE del procesador que la ejecuta. El almacenamiento en los BE se efectúa de forma paralela. El siguiente acceso, de cada uno de los procesadores, es una lectura. El procesador P1 lee la variable aviso2 y el procesador P2 lee la variable aviso1. Las lecturas tienen prioridad respecto de la cabeza del BE. Entonces, las lecturas obtienen el bus.

49. El tipo de red utilizada (bus) permite que la respuesta pueda considerarse implícita al obtener acceso al bus.
50. En un sistema uniprocesador es necesario vaciar el BE en un cambio de contexto del procesador. Pensemos que aunque se utilicen direcciones físicas, el SO puede modificar la correspondencia entre el espacio lógico y el espacio físico.

El bus está ocupado durante todo el acceso a memoria. Entonces, los accesos se efectúan de forma serie. El valor leído de las variables aviso1 y aviso2 es cero. En consecuencia no se mantiene consistencia secuencial. Después de las lecturas se actualiza memoria con la información del BE[51].

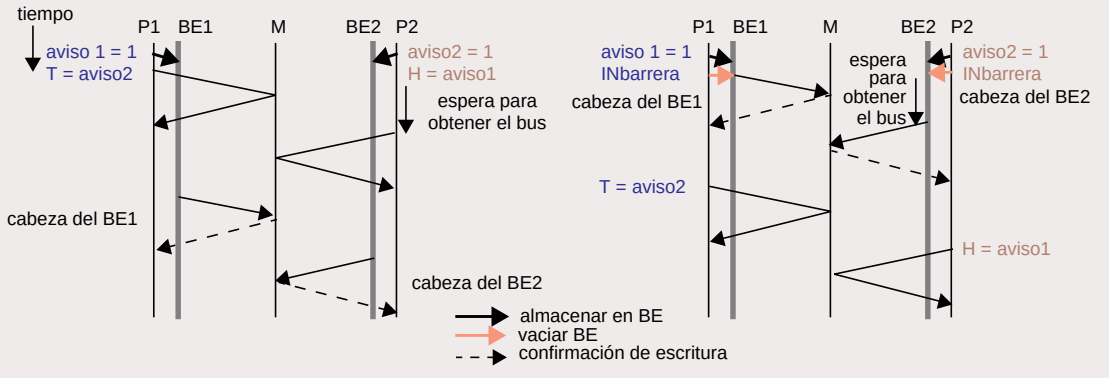


Figura 3.31 Multiprocesador con procesadores que utilizan un BE. Diagrama temporal de accesos a memoria.

En la parte derecha de la Figura 3.31 se muestra el diagrama temporal de los accesos a memoria al ejecutar el código de la parte derecha de la Figura 3.30 con las instrucciones INbarrera[52].

La microarquitectura utiliza la instrucción INbarrera para consolidar todos los accesos a memoria pendientes y no iniciar ningún acceso posterior hasta que se haya consolidado la instrucción INbarrera. Esto es, el BE debe vaciarse. Una vez consolidada la última entrada ocupada del BE se consolida la ejecución de la instrucción INbarrera. Entonces, el procesador puede ejecutar la siguiente instrucción de acceso a memoria. El valor leído de las variables aviso1 y aviso2 es uno. En consecuencia se mantiene consistencia secuencial.

Coherencia de cache

Respecto a coherencia de cache, la condición de atomicidad de las escrituras puede relajarse. Un procesador puede utilizar el valor de una escritura, antes de ser consolidada (propagada), en una instrucción load más joven. La coherencia de cache se mantiene, ya que otros procesadores no observan el valor. La lectura se puede ordenar, en el orden de accesos a la posición de memoria, inmediatamente después de que la escritura consolide[53] y el sistema es coherente.

51. Notemos que las escrituras se consolidan y lecturas posteriores leen el valor establecido por estas escrituras.
52. Notemos que la acción de una instrucción INbarrera es local. Establece ordenación de accesos a memoria en un hilo.

En estas condiciones, la condición de atomicidad de escrituras se relaja a la condición de serialización de escrituras, la cual es subyacente en la condición de atomicidad.

Serialización de escrituras. Todas las escrituras a una posición de memoria son observadas en el mismo orden por todos los procesadores.

Notemos que un procesador, después de ejecutar una instrucción store, puede ejecutar inmediatamente una instrucción load, a la misma posición de memoria, sin tener que esperar la confirmación de la escritura, es suficiente garantizar que la instrucción store y la instrucción load se ejecutan en orden de programa[54].

Ejemplo. Suponemos un multiprocesador que utiliza un bus como red de interconexión y cada procesador tiene un BE y una cache privada. Las cache privadas utilizan escritura inmediata y el protocolo de coherencia es de tipo invalidación (Figura 3.32).

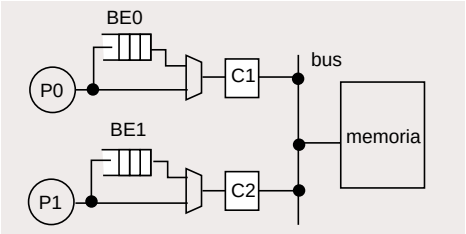


Figura 3.32 Multiprocesador con procesadores que utilizan un BE y tienen cache privada.

En la Figura 3.33 se muestra un ejemplo de código y su ejecución en el multiprocesador de la Figura 3.32. Los procesadores almacenan la instrucción store en el BE. Cuando ejecutan las instrucciones load obtienen el valor del BE correspondiente. En paralelo se propagan las escrituras.

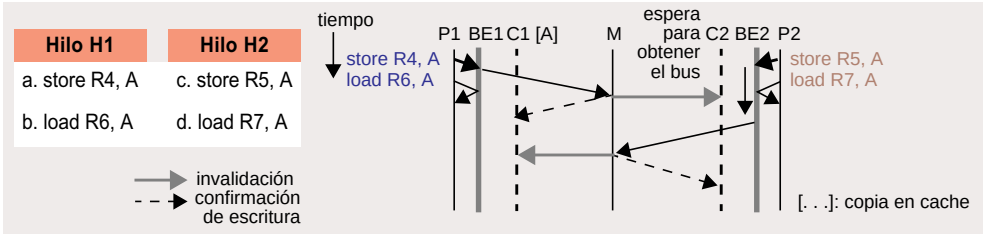


Figura 3.33 Serialización de escrituras.

53. Desde el punto de vista de coherencia, esta relajación permite la utilización de BE o que una cache se actualice antes de recibir la confirmación de la consolidación.
54. Es una dependencia de datos del hilo. Por ejemplo, en un multiprocesador donde los procesadores no tienen BE y el camino a memoria de las instrucciones store y de las instrucciones load es distinto hay que garantizar el orden. Para ello, la instrucción load debe esperar la confirmación de la escritura.

Al utilizarse un bus como red de interconexión las caches observan la transacción e invalidan la copia en cache. La cache C2 invalida la copia al propagarse la escritura del procesador P1. La cache C1 se actualiza al recibir la confirmación de la escritura.

La cache C1 invalida la copia al propagarse la escritura del procesador P2. La cache C2 no se actualiza al recibir la confirmación de la escritura, ya que el bloque está invalidado. Al finalizar la ejecución, la cache no tienen copia del bloque y memoria almacena el valor establecido por la escritura del procesador P2. El orden de los accesos a la posición de memoria A es: a, b, c, d.

Condiciones de coherencia de cache. A partir de ahora, las condiciones para coherencia de cache son: a) propagación de escritura y b) serialización de escrituras, en lugar de atomicidad de escrituras.

En estas condiciones, mediante el mecanismo de coherencia no se conoce el instante a partir del cual una escritura es observable por otros procesadores. El procesador que ejecuta una instrucción load puede obtener el valor de un entorno local como un BE o una cache.

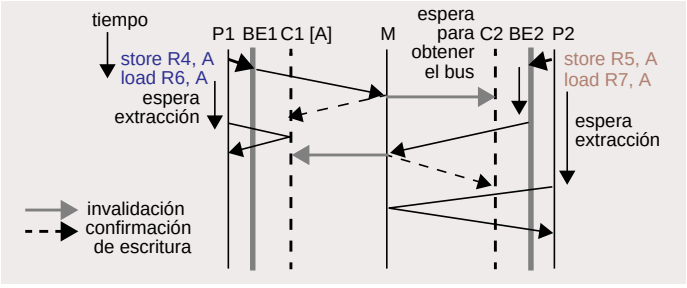
Ejercicio

Suponga que el BE dispone de lógica para detectar dependencias, pero no dispone de lógica para suministrar el dato. Razone la actuación que debe tomarse en caso de detectar una dependencia. Utilice el código de la Figura 3.33 para mostrar en un diagrama temporal el funcionamiento en el multiprocesador de la Figura 3.32*. Suponga que el orden temporal de ejecución de los accesos a memoria es: a, c, b, d.*

Respuesta

La lógica debe bloquear la instrucción load hasta que en el BE no exista una dirección coincidente. Una instrucción load obtendrá el valor de la escritura emitida por el mismo procesador o de una escritura emitida por otro procesador, en función de la serialización en el entrelazado de accesos a la misma posición de memoria.

En la siguiente figura se muestra el diagrama temporal.



La ejecución de la instrucción load espera a que se extraiga del BE la instrucción store con dirección coincidente. En el procesador P1 la instrucción load es acierto en cache. En el procesador P2 la instrucción load es fallo de cache, ya que la copia ha sido invalidada por la escritura P1.

EJEMPLOS

Orden de programa: compilador

Para mantener consistencia secuencial es necesario mantener los siguientes ordenes de programa en instrucciones de acceso a memoria que acceden a posiciones distintas de memoria.

orden de programa ↓			
lectura	lectura	escritura	escritura
lectura	escritura	lectura	escritura

En los trozos de código de cada pregunta razone sobre los ordenes que deben mantenerse.

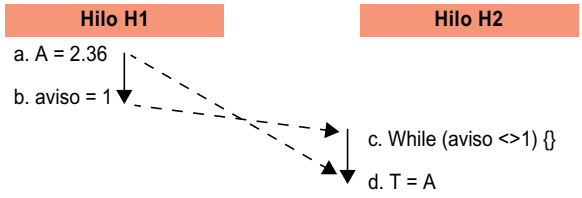
Pregunta 1: *Sincronización punto a punto mediante eventos. La variable aviso ha sido inicializada a cero.*

Hilo H1	Hilo H2
A = 2.36	While (aviso <>1) {}
aviso = 1	T = A

Respuesta: Deben mantenerse los siguientes dos ordenes.

lectura	escritura
lectura	escritura

El primero de ellos en el hilo H2 y el segundo en el hilo H1. El orden de programa determina a -> b (E-E) y c -> d (L-L). La ordenación b -> c implica la ordenación a -> d.



Pregunta 2: *Sincronización de tres hilos mediante eventos. Las variables A y B han sido inicializadas a cero.*

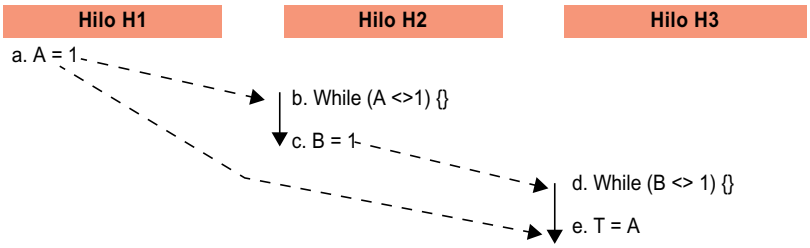
Hilo H1	Hilo H2	Hilo H3
A = 1	While (A <>1) {}	While (B <> 1) {}
	B = 1	T = A

Respuesta: Debe mantenerse el siguiente orden.

lectura

escritura

Los ordenes de programa determinan b -> c (L-E) y d -> e (L-E). La ordenación a -> b implica la ordenación a -> c. La ordenación c -> d implica la ordenación b -> c -> e. Por tanto se cumple la ordenación a -> e.



Pregunta 3: *Algoritmo de Dekker para exclusión mutua. Las variables aviso1 y aviso2 han sido inicializadas a cero. La variable turno ha sido inicializada a uno.*

	Hilo H1	Hilo H2
petición	aviso1 = 1	aviso2 = 1
pregunta	While (aviso2 = 1) {	While (aviso1 = 1) {
decisión si hay	While (turno <> 1) {	While (turno <> 2) {
competencia	aviso1 = 0	aviso2 = 0
	}	}
petición	aviso1=1	aviso2=1
	}	}
exclusión	Acceso exclusivo	Acceso exclusivo
cambio de turno	turno =2	turno = 1
liberación	aviso1 = 0	aviso2 = 0

En este algoritmo se pone a uno una variable (aviso1 o aviso2), para indicar que el hilo quiere conseguir acceso exclusivo. Un hilo, antes de entrar, comprueba si el otro hilo también pretende entrar. En caso de que observe que el otro hilo quiere entrar y no sea su turno declina de su pretensión y lo vuelve a intentar posteriormente.

Respuesta: En el algoritmo de Dekker debe mantenerse el orden.

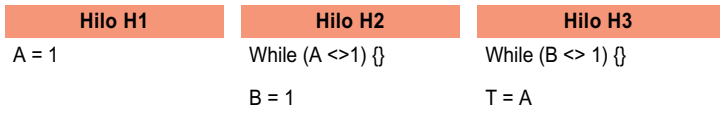
escritura

lectura

El orden de programa determina a -> b (E-L) y c -> d (E-L). Las ordenaciones a -> d y c -> b implican tomar un camino u otro después de d o b.

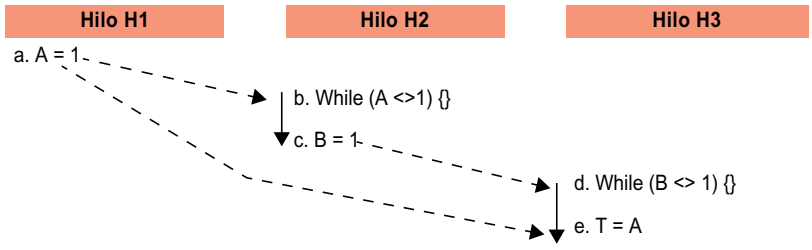
Atomicidad de las escrituras

Sincronización de tres hilos mediante eventos. Las variables A y B han sido inicializadas a cero.



Pregunta 1: *Justifique la necesidad de atomicidad en las escrituras utilizando el ejemplo.*

Respuesta: Los ordenes de programa determinan b -> c y d -> e. La ordenación a -> b implica la ordenación a -> c. La ordenación c -> d implica la ordenación b -> c -> e. Por tanto se cumple la ordenación a -> b -> e. Las sentencias b y e leen la misma variable. Entonces, el valor debe ser el mismo.



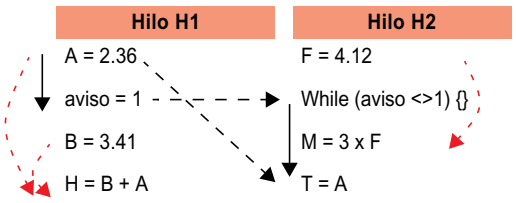
Semántica que espera el programador

Muestre, utilizando los siguientes trozos de código, que las condiciones enumeradas para disponer de consistencia secuencial son más restrictivas que las necesarias para mantener la semántica que espera el programador. Para ello, razone sobre si se pueden reescribir los trozos de código y seguir efectuando el mismo cálculo.

Pregunta 1: *Las variables B, H, F, T Y M están en el espacio de direcciones compartido pero son accedidas exclusivamente por uno sólo de los procesos.*

Hilo H1	Hilo H2
B = 3.41	F = 4.12
A = 2.36	While (flag <>1) {}
flag = 1	T = A
H = B + A	M = 3 x F

Respuesta: Las únicas variables involucradas en la comunicación y sincronización entre hilos son las variables A y aviso. Las otras variables no se comparten. Por tanto, se puede modificar el orden mientras se respeten las dependencias de datos dentro de cada hilo. Una posible ordenación que efectúa el mismo cálculo es la siguiente.



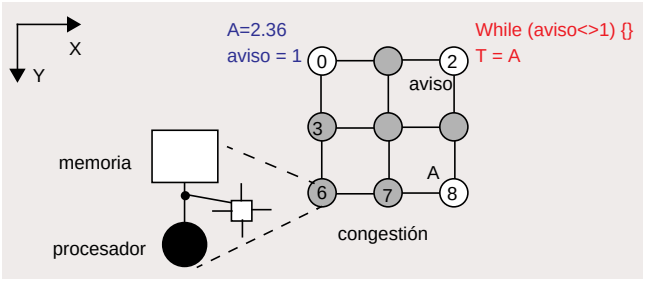
Las variables T y M pueden considerarse locales.

Por tanto, las condiciones enumeradas para consistencia secuencial son suficientes, pero no son necesarias para mantener la semántica que espera el programador.

Tiempo de ejecución

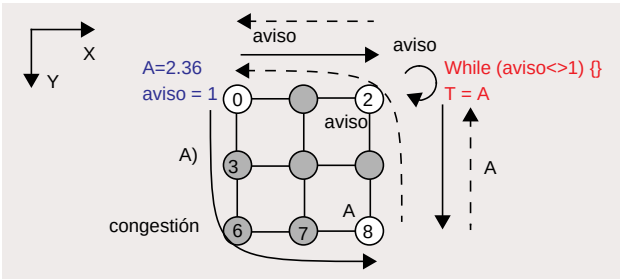
Suponga un sistema multiprocesador donde la red de interconexión es una malla. Un acceso se encamina en primer lugar siguiendo la dirección Y y posteriormente se sigue por la dirección X.

En los procesadores de los nodos cero y 2 se ejecutan los trozos de código mostrados. Los dos hilos pertenecen a un programa paralelo. Suponga que el acceso a una memoria local tarda 1 ciclo y que, al efectuar un acceso a un módulo de memoria no local, la transmisión de información entre dos nodos adyacentes también tarda 1 ciclo. La confirmación en una instrucción store tarda en generarse 1 ciclo y se efectúa en paralelo con la actualización de memoria. Suponga que en el camino de ida, en el acceso a memoria relativo a la variable A, hay congestión y se tardan 2 ciclos más de lo imprescindible.



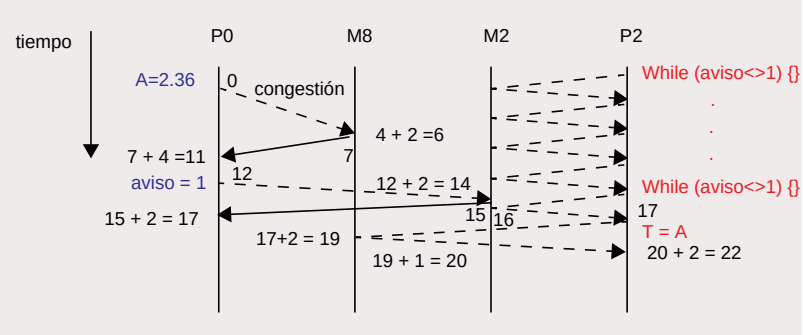
Pregunta 1: *Calcule el tiempo de ejecución.*

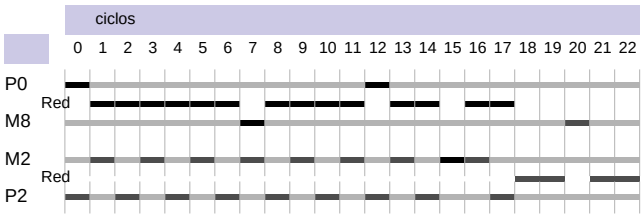
Respuesta: En el siguiente diagrama se muestran el recorrido de los accesos en la ida y en la vuelta. La línea continua indica ida y la línea discontinua indica vuelta.



En la siguiente figura se muestra el cálculo de los tiempos.

En el camino de ida de la instrucción store A se tardan 4 saltos (4 comunicaciones entre nodos adyacentes) más 2 ciclos debidos a la congestión del camino. Por tanto, el acceso llega a la memoria del nodo 8 en el ciclo 6, si empezamos a contar desde el ciclo 0. En el camino de vuelta no se encuentra congestión y son 4 saltos, lo que equivale a 4 ciclos. Entonces la confirmación llega al procesador en el ciclo 11, ya que ha salido en el ciclo 7.





El acceso a la variable aviso (store aviso) se envia en el ciclo 12. Este acceso llega a la memoria del nodo 2 en el ciclo 14 y la confirmación llega al procesador en el ciclo 17 (15 + 2).

Suponemos que la variable aviso se lee en el ciclo 16, ya que el acceso de actualización de esta variable ha llegado a memoria en el ciclo 14 y el ciclo 15 se utiliza para actualizar memoria. Como es un acceso a memoria local el procesador efectúa el siguiente acceso (load A) en el siguiente ciclo (17). Este acceso llega a la memoria del nodo 8 en el ciclo 19 y después de leer (1 ciclo) se envía el dato al procesador. El dato llega al procesador en el ciclo 22.

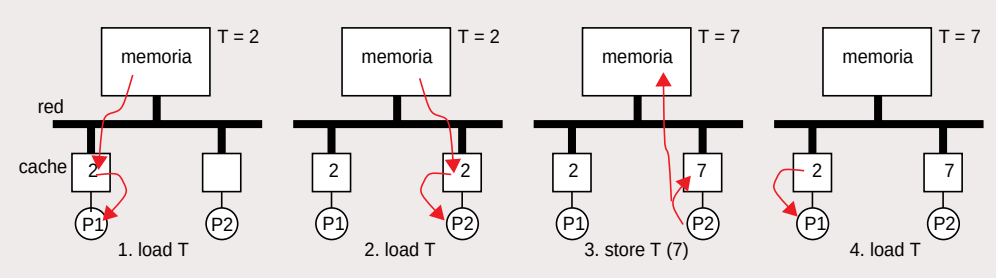
Coherencia de cache

Un multiprocesador dispone de dos procesadores. Las caches privadas utilizan escritura inmediata para mantener la coherencia. Suponga que se está ejecutando un proceso que migra entre los procesadores.

Pregunta 1: *Muestre que en este multiprocesador los resultados que se calculan pueden ser incorrectos.*

Respuesta: Supongamos que la variable T no está almacenada en ninguna cache inicialmente. El proceso lee la variable T cuando se ejecuta en el procesador P1 y se produce un fallo de cache. El bloque que contiene la variable se trae a un contenedor de su cache. Posteriormente el proceso migra al procesador P2 y lee la variable T. Se produce un fallo y el bloque que contiene la variable se almacena en un contenedor de su cache. Ahora hay 2 copias del bloque en las caches y son idénticas. Seguidamente, mientras el proceso se ejecuta en el procesador P2 se escribe la variable T. Como se utiliza escritura inmediata se actualiza la cache y memoria. Ahora las copias ya no son idénticas.

Después de un intervalo de tiempo el proceso migra al procesador P1 y lee la variable T. La variable T está almacenada en un contenedor de su cache y por tanto es un acierto. Ahora bien, el valor de la variable T no es el último valor que ha escrito el proceso en la variable T. Este valor está almacenado en la copia residente en la cache del procesador P2 o en memoria. Por tanto, la ejecución será incorrecta, ya que se lee un valor antiguo.



En la siguiente tabla se muestra en cada fila una fotografía de las caches y memoria en cada acceso a memoria. Las filas están en orden temporal de arriba a abajo, representado en la primera fila el primer acceso.

Actividad del procesador	Actividad de la red	Cache de P1	Cache de P2	Memoria	Coherencia en copias
				2	
1. P1 load T	fallo	2	no hay copia	2	coherente
2. P2 load T	fallo	2	2	2	coherente
3. P2 store T	acierto	2	7	7	incoherente
4. P1 load T	acierto	2	7	7	incoherente

APENDICE A: SISTEMA UNIPROCESADOR Y ENTRADA/SALIDA

El problema de coherencia de cache también existe, en sistemas uniprocesador, cuando se efectúa comunicación con el exterior. Las transferencias de datos, debidas a la entrada/salida de información, se efectúan mediante canales de acceso directo a memoria (direct memory access, DMA) o procesadores especializados de entrada/salida (Figura 3.34). Estos dispositivos mueven datos entre la memoria y los dispositivos exteriores, en ambos sentidos, sin intervención del procesador. Cuando un canal de DMA escribe en memoria el procesador puede seguir leyendo un valor antiguo (más viejo), almacenado en cache si no se efectúa ninguna acción. Igualmente, cuando un canal de DMA lee de memoria puede leer un valor antiguo si, se utiliza escritura retardada para mantener la coherencia en la jerarquía y el procesador ha actualizado el dato en cache.

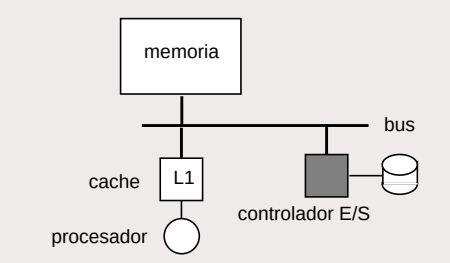


Figura 3.34 Sistema uniprocesador con comunicación con el exterior.

Las operaciones de entrada/salida son menos frecuentes que los accesos a memoria mediante instrucciones load y store desde el procesador. Entonces, pueden adoptarse soluciones más groseras que en un multiprocesador. Por ejemplo, páginas de memoria utilizadas por la entrada/salida se pueden marcar como no cacheables (los bloques de memoria que contienen no se almacenan en cache). También se pueden utilizar instrucciones load y store marcadas de forma que el dato accedido no se almacena en cache. Estas instrucciones se utilizan al comunicarse con los dispositivos de entradas/salida para indicar las operaciones o comprobar si se ha producido algún error. Otra posibilidad para mejorar la eficiencia es involucrar al sistema operativo. El sistema operativo elimina de la cache de los procesadores, usualmente mediante instrucciones específicas, los bloques de las páginas de memoria sobre las que se efectúa la operación de entrada/salida, antes de que se efectúe propiamente la operación de entrada/salida.

EJERCICIOS

Ejercicio 3.1

Considere la ejecución de 2 programas por 2 procesadores con memoria compartida. Suponga que las variables A, B, C, D se inicializan a cero y que la sentencia PRINT imprime los dos argumentos en el mismo ciclo de forma indivisible. Las salidas son 4-tuplas que se obtienen al ejecutar las sentencias c y f. El primer par de valores de la 4-tupla se corresponde con la sentencia que se ejecuta en primer lugar.

P0	P1
a. A = 1	d. C = 1
b. B = 1	e. D = 1
c. PRINT A, D	f. PRINT B,C

Pregunta 1: *Enumere todos los ordenes de ejecuciones entrelazadas de las 6 sentencias que preservan el orden del programa.*

Pregunta 2: *Suponga que el orden del programa se preserva y todos los accesos a memoria son atómicos. Es decir, un store efectuado por un procesador es observado inmediatamente por todos los procesadores restantes. Enumere todas las posibles 4-tuplas de salida.*

Pregunta 3: *Suponga que el orden del programa se preserva pero los accesos a memoria no son atómicos; es decir un store efectuado por un procesador se almacena en un buffer de forma que otros procesadores no pueden observar inmediatamente la actualización. Enumere todas las posibles 4-tuplas de salida.*

Ejercicio 3.2

Considere la ejecución de 3 programas por 3 procesadores con memoria compartida. Suponga que las variables A, B, C se inicializan a cero y que la sentencia PRINT imprime los dos argumentos en el mismo ciclo de forma indivisible. Las salidas son 6-tuplas de vectores de bits, que se obtienen al ejecutar las sentencias b, d y f. El primer par de valores de la 6-tupla se corresponde con la sentencia que se ejecuta en primer lugar y el segundo par de valores con la que se ejecuta en segundo lugar.

P0	P1	P2
a. A =1	c. B=1	e. C = 1
b. PRINT B, C	d. PRINT A,C	f. PRINT A, B

Pregunta 1: *Suponga que todos los procesadores ejecutan las instrucciones en el orden que especifica su programa. Determine que las salidas 001011 y 111111, pertenecen a una ejecución de los programas.*

Pregunta 2: *Suponga que todos los procesadores ejecutan las instrucciones en el orden que especifica su programa. Determine que la salida 000000 no es posible.*

Pregunta 3: *Suponga que todos los procesadores ejecutan las instrucciones en el orden que especifica su programa. Determine que la salida 011001 es posible si procesadores distintos observan los eventos en ordenes distintos.*

Ejercicio 3.3

El siguiente código implementa el algoritmo de Dekker para la exclusión mutua entre dos procesos.

P0	P1	valores iniciales
A =1	B =1	A = B = 0
while (B =1) {. . .};	while (A =1) {. . .};	
. . .	. . .	
R.Critica	R.Critica	
. . .	. . .	
A = 0	B = 0	

Una simplificación del código para efectuar un análisis de consistencia secuencial es

P0	P1	valores iniciales
1. A =1	3. B = 1	A = B = 0
2. x = B	4. y = A	

Pregunta 1: *Muestre que si se cumple consistencia secuencial las variables (x, y) pueden tomar los valores (1,0), (0,1) o (1,1) pero no (0,0).*

Ejercicio 3.4

El siguiente código utiliza una instrucción que efecua las operaciones leer-modificar-escribir de forma indivisible.

P0	P1	valores iniciales
1.1) x = A	2. A = 1	A = 0
1.2) A = x + 2	3. y = A	

Las instrucciones 1.1) y 1.2) se ejecutan de forma indivisible.

Pregunta 1: *Muestre que si se cumple consistencia secuencial las variables (x, y) pueden tomar los valores (0,1), (1,1) o (1,3) pero no (0,2).*

Ejercicio 3.5

Un hilo productor y un hilo consumidor utilizan un buffer circular para comunicarse información. Cada uno de ellos se ejecuta en un procesador distinto. El hilo productor inserta un dato en un buffer compartido si el buffer no esta lleno. El hilo consumidor extrae un dato del buffer si no esta vacio.

El buffer tiene N posiciones de almacenamiento y se implementa mediante un vector compartido y tres variables compartidas denominadas cola, cabeza y cont, las cuales indican respectivamente la próximas posiciones de inclusión y extracción y el número de posiciones ocupadas del buffer.

Seguidamente se muestran tres versiones del trozo de código que nos interesa. La actualización de la variable cont se efectúa de forma atómica mediante una instrucción. En todas las versiones, las variables están inicializadas a cero:

A)	productor	consumidor
	while (cont = N) {};	While (cont = 0) {};
	buffer(cola) = item;	item = buffer(cabeza);
	cola = (cola + 1) mod N;	cabeza = (cabeza + 1) mod N;
	cont = cont +1;	cont = cont - 1;

B)	productor	consumidor
	while (cont = N) {};	While (cont = 0) {};
	cont = cont +1;	cont = cont - 1;
	buffer(cola) = item;	item = buffer(cabeza);
	cola = (cola + 1) mod N;	cabeza = (cabeza + 1) mod N;

C)	productor	consumidor
	while (cont = N) {};	While (cont = 0) {};
	buffer(cola) = item;	item = buffer(cabeza);
	cont = cont +1;	cont = cont - 1;
	cola = (cola + 1) mod N;	cabeza = (cabeza + 1) mod N;

Suponga que tanto el compilador como el hardware garantizan consistencia secuencial.

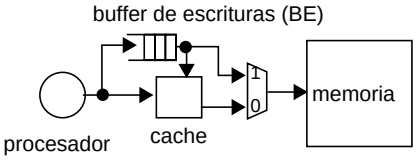
Pregunta 1: *Indique la versiones que funcionan de forma correcta y las que no lo hacen. Justifique la respuesta*

Suponga que el compilador o el hardware no garantizan consistencia secuencial. La necesidad de garantizar consistencia secuencial se puede efectuar mediante la inserción de la directiva LNbarrera.

Pregunta 2: *En las versiones que ha identificado que funcionan correctamente indique la posición en el código en la cual debe incluirse la directiva LNbarrera. Justifique la respuesta.*

Ejercicio 3.6

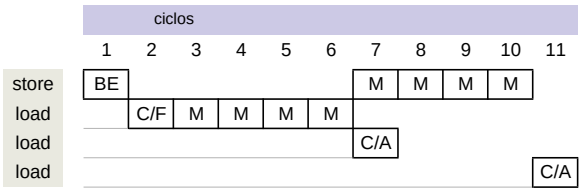
Un procesador utiliza una cache con escritura inmediata y sin asignación de contenedor en un fallo de escritura. Para que las escrituras no bloqueen al procesador se utiliza un buffer denominado de escrituras (BE). En la siguiente figura se muestra el camino de datos cuando se produce un fallo de lectura o una escritura (acierto o fallo). En una escritura la cache se actualiza si se produce un acierto.



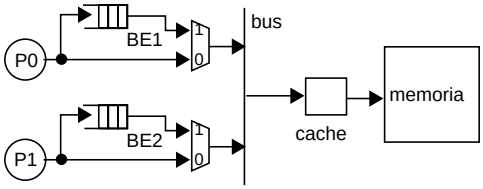
El dato en una instrucción store se almacena en el BE y el procesador sigue interpretando instrucciones (store no bloqueante). En una entrada del BE se distinguen los campos dirección y dato. Una operación store actualiza memoria cuando el camino de acceso está libre. La gestión del BE es FIFO.

Para mejorar el rendimiento se permite que instrucciones load más jóvenes adelanten a operaciones store almacenada en el BE. La justificación es que el procesador necesita los datos para efectuar cálculos y proseguir interpretando instrucciones. En cambio, la actualización determinada por una instrucción store se puede retardar. En el caso de que la dirección a la que accede una instrucción load más joven coincida con una dirección en el BE el dato se suministra desde el BE (esta parte del camino de datos no se muestra en la figura previa).

Si el buffer de escrituras está lleno y se inicia la interpretación de una nueva instrucción store se bloque la interpretación de instrucciones hasta que se ha vaciado el BE. En la figura se muestra un ejemplo donde se observa como una instrucción load adelanta a una instrucción store más vieja. Los acrónimos C, A, F y M indican respectivamente acceso a cache, acierto en cache, fallo en cache y acceso a memoria.



Este procesador se utiliza en el diseño de un multiprocesador que utiliza un bus como red de interconexión y la cache de primer nivel se comparte por todos los procesadores. En estas condiciones en cada procesador pueden competir por el bus la entrada en la cabeza del BE y una instrucción load. En este caso el fallo es prioritario. En la siguiente figura se muestra un esquema de un multiprocesador con dos procesadores.



En este multiprocesador se ejecutan los siguientes dos trozos de código (esqueleto incompleto) que implementan el algoritmo de Dekker para acceso exclusivo.

P0	P1	valores iniciales
aviso1 =1	aviso2 =1	aviso1 = aviso2 = 0
while (aviso2 =1) {...};	while (aviso1 =1) {...};	
. . .	. . .	
R.Critica	R.Critica	
. . .	. . .	
aviso1 = 0	aviso2 = 0	

Una simplificación del código para efectuar un análisis de consistencia secuencial es

P0	P1	valores iniciales
1. aviso1 = 1	3. aviso2 = 1	aviso1 = aviso2 = 0
2. x = aviso2	4. y = aviso1	

Inicialmente, en contenedores de cache están almacenados los bloques que contienen las variables aviso1 y aviso2.

Pregunta 1: *Muestre que este multiprocesador no cumple las condiciones de consistencia secuencial al ejecutarse los códigos previos.*

La necesidad de garantizar consistencia secuencial se puede efectuar mediante la inserción de la directiva LNbarrera.

Pregunta 2: *Indique las posiciones en el código en las cuales debe incluirse la directiva LNbarrera. Justifique la respuesta.*

Un segundo código que se ejecuta en el multiprocesador es el siguiente.

Hilo H1	Hilo H2
A = 2.36	While (aviso <>1) {}
aviso = 1	T = A

Inicialmente los bloques que contienen las variables están almacenados en cache. El valor de los variables es cero.

Pregunta 3: *Justifique si se cumple consistencia secuencial.*

Un cuarto código que se ejecuta en el multiprocesador es el siguiente.

Hilo H1	Hilo H2	Hilo H3
A = 1	While (A <>1) {}	While (B <> 1) {}
	B = 1	T = A

Inicialmente los bloques que contienen las variables están almacenados en cache. El valor de los variables es cero.

Pregunta 4: *Justifique si se cumple consistencia secuencial.*

Finalmente, otro código que se ejecuta en el multiprocesador es el siguiente.

P0	P1	valores iniciales
1. aviso1 = 1	4. aviso2 = 1	aviso1 = aviso2 = 0
2. P = aviso1	5. Q = aviso2	
3. T = aviso2	6. V = aviso1	

Inicialmente los bloques que contienen las variables están almacenados en cache.

Pregunta 5: *Indique los valores que almacenan las variables P, T, Q y V.*

Ejercicio 3.7

En un multiprocesador se permite migración de procesos entre procesadores. Este multiprocesador no dispone de un mecanismo para mantener la coherencia entre cache privadas. Suponga escritura inmediata en las caches privadas.

Pregunta 1: *Muestre que puede producirse un problema de coherencia de cache al ejecutar un programa secuencial. Para ello utilice una traza de accesos a memoria.*

Ejercicio 3.8

Un hilo productor y un hilo consumidor utilizan un buffer circular para comunicarse información. Cada uno de ellos se ejecuta en un procesador distinto. El hilo productor inserta un dato en un buffer compartido si el buffer no esta lleno. El hilo consumidor extrae un dato del buffer si no esta vacio.

El buffer tiene N posiciones de almacenamiento y se implementa mediante un vector compartido y tres variables compartidas denominadas cola, cabeza y cont, las cuales indican respectivamente las posiciones de inclusión y extracción y el número de posiciones ocupadas del buffer.

El código relevante del productor y del consumidor se muestra seguidamente. La variable cont se actualiza mediante una instrucción atómica. Todas las variables están inicializadas a cero.

productor (P)	consumidor (C)
while (cont = N) {};	While (cont = 0) {};
buffer(cola) = item;	item = buffer(cabeza);
cola = (cola + 1) mod N;	cabeza = (cabeza + 1) mod N;
cont = cont +1;	cont = cont - 1;

Suponga que los hilos P y C insertan o extraen K veces consecutivas elementos del buffer. Es decir, el hilo P inserta K elementos y el hilo C no extrae ningún elemento. Posteriormente, el hilo C extrae K elementos y el hilo P no inserta ningún elemento.

Suponga que el tamaño de bloque es exactamente igual a un elemento del buffer, que las variables compartidas ocupan un bloque cada una de ellas y que el tamaño de la cache es infinito. También suponga que todas las posiciones del buffer han sido accedidas al menos una vez por el hilo productor y el hilo consumidor.

Pregunta 1: *Considere un esquema de coherencia de cache con política de invalidación en las escrituras. Calcule el número de fallos de cache y el número de invalidaciones en función del valor de K.*

Pregunta 2: *Considere un esquema de coherencia de cache con política de actualización en las escrituras. Calcule el número de fallos de cache y el número de actualizaciones en función del valor de K.*

Pregunta 3: *Razone, en función de K, el efecto que tiene el tamaño de bloque sobre el número de fallos, número de invalidaciones y el tráfico de bus.*

Ejercicio 3.9

Un algoritmo iterativo para resolver sistemas lineales de ecuaciones es el siguiente.

productor

```
repeat
    X_{i+1} = A X_i + B
until (. . . .)
```

donde X_{i+1}, X_i y B son vectores de tamaño N y A es una matriz de tamaño N x N. El calculo finaliza (acaba de iterar) cuando la diferencia entre X_{i+1} y X_i es menor que un valor predeterminado. La condición de terminación no se ha incluido ya que no es importante para el propósito del problema.

Suponga que cada iteración (X_{i+1}) se calcula utilizando N procesos, donde cada proceso calcula un elemento del vector. El código utilizado es el siguiente.

productor

```
repeat
    doall J =1 , N
         xtemp(J) = B(J)
        do K =1, N
            xtemp(J) = xtemp(J) + A(J,K) * X(K)
        endo
    endoall
    BARRERA
    doall J =1, N
        X(J) = xtemp(J)
    endoall
    BARRERA
until false
```

El bucle doall inicia N procesos. Cada proceso calcula un nuevo valor, el cual se almacena en xtemp. El segundo bucle paralelo copia los elementos del vector xtemp en el vector x. Esta operación requiere una sincronización tipo BARRERA.

Observe que los elementos del vector B y la matriz A solo se leen y los elementos de los vectores X y xtemp se leen y escriben

Suponga que el tamaño del bloque es exactamente igual a un elemento de la matriz o de los vectores y que el tamaño de cache es infinito.

Pregunta 1: *Considere un esquema de coherencia de cache con política de invalidación en las escrituras. Calcule, en el peor caso, el número de fallos de cache y el número de invalidaciones para cada elemento de las estructuras de datos del programa.*

Pregunta 2: *Considere un esquema de coherencia de cache con política de actualización en las escrituras. Calcule el número de fallos de cache y el número de actualizaciones para cada elemento de las estructuras de datos del programa.*

Pregunta 3: *Razone el efecto que tiene el tamaño de bloque sobre la frecuencia de fallo y el tráfico de bus.*

Suponga que el compilador o el hardware no garantizar consistencia secuencial. La necesidad de garantizar consistencia secuencial se puede efectuar mediante la inserción de la directiva LNbarrera.

Pregunta 4: *Indique las posiciones en el código en las cuales debe incluirse la directiva LNbarrera. Justifique la respuesta.*

Ejercicio 3.10

"Seqlock" es un mecanismo de sincronización que permite que un proceso actualice variables compartidas (proceso actualizador) y otros procesos lean estas variables (procesos lectores). El proceso actualizador nunca se espera para efectuar la actualización, mientras que los procesos lectores tiene que esperar si el proceso actualizador está accediendo a las variables.

El mecanismo "seqlock" utiliza una variable compartida para almacenar un número de secuencia (numsec). La variable numsec es actualizada por el proceso actualizador para indicar que está actualizando las variables compartidas. Los procesos lectores utilizan la variable numsec para determinar si los datos leídos de las variables compartidas son consistentes.

El proceso actualizador incrementa la variable numsec antes y después de actualizar las variables compartidas.

Un proceso lector lee el número de secuencia (numsec). Si el número de secuencia es impar el proceso actualizador está modificando las variables compartidas. Si los números de secuencia leídos por el proceso lector antes y después de leer las variables compartidas son distintos, entonces el proceso actualizador ha modificado los datos compartidos mientras se estaban leyendo. El proceso lector debe de volver a efectuar las acciones de leer las variables

compartidas y el número de secuencia antes y después de leer las variables compartidas hasta que los números de secuencia leídos, que deben ser par, sean el mismo.

Los códigos del proceso actualizador y de un proceso lector son los siguientes:

proceso actualizador	proceso lector	inicialización
numsec = numsec +1	repeat	numsec = 0
dato1 = . . .	repeat	numsecini es local
dato2 = ...	numsecini = numsec	
numsec = numsec +1	until (numsecini and 1 = 0)	
	. . . = dato1	
	. . . = dato 2	
	until (numsecini = numsec)	

Pregunta 1: *Suponga que los códigos previos se ejecutan en un sistema multi-procesador donde se garantiza consistencia secuencial (compilador, hardware). Indique si las dos siguientes versiones de los procesos actualizador y lector producen los mismos resultados que el código previo. Justifique la respuesta.*

VERSION A			VERSION B	
proceso actualizador	proceso lector	inicialización	proceso actualizador	proceso lector
numsec = numsec +1	repeat	numsec = 0	numsec = numsec +1	repeat
dato2 = . . .	repeat		dato2 = . . .	repeat
dato1 = ...	numsecini = numsec		dato1 = ...	numsecini = numsec
numsec = numsec +1	until (numsecini and 1 = 0)		numsec = numsec +1	until (numsecini and 1 = 0)
	. . . = dato1			. . . = dato2
	. . . = dato2			. . . = dato1
	until (numsecini = numsec)			until (numsecini = numsec)

Los códigos previos se ejecutan en un sistema multiprocesador que no garantiza consistencia secuencial. Para garantizarla utilizaremos la directiva LNbarrera. Al insertar esta directiva entre dos sentencias se inhibe en el compilador la posibilidad de reordenar operaciones de acceso a memoria. El compilador no puede ubicar después de la directiva LNbarrera operaciones de acceso a memoria que el programador ha especificado antes de la directiva LNbarrera y viceversa. Desde el punto de vista hardware la directiva se traduce en una instrucción INbarrera que garantiza que las operaciones de acceso a memoria, previas a la instrucción INbarrera, han sido consolidadas antes de iniciar la ejecución de las operaciones de acceso a memoria especificadas después de la instrucción INbarrera.

Pregunta 2: *Indique las posiciones en el código en las cuales hay que insertar la directiva LNbarrera para garantizar consistencia secuencial. Justifique la respuesta. El número de inserciones de la directiva LNbarrera debe ser el mínimo posible.*