5

LA MEMORIA EN LOS SISTEMAS PARALELOS

5.1. Introducción

En este capítulo nos ocuparemos del diseño de memorias para sistemas paralelos débil y fuertemente acoplados. Por otra parte, también abordaremos algunos problemas planteados por las memorias compartidas. El más importante de estos problemas es el de la **coherencia caché** que se refiere a la dificultad para mantener actualizadas todas las memorias caché de los procesadores cuando tienen la memoria principal compartida.

5.2. Organizaciones de memoria para los multiprocesadores

Entre las diferentes organizaciones de sistemas que podemos tener, se encuentran principalmente tres:

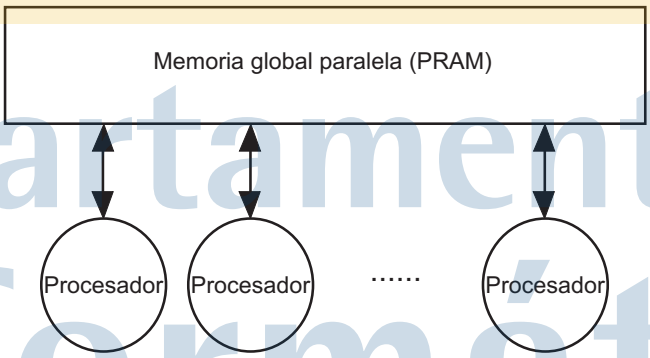

Fig. 5.1. Modelo PRAM teórico.

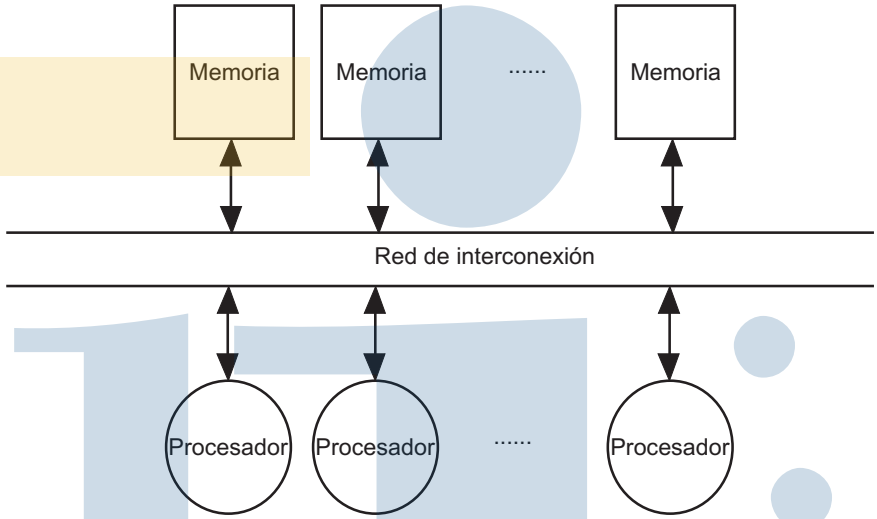

Fig. 5.2. Multiprocesador con acceso a memoria uniforme (UMA).

UMA (***uniform memory access*, acceso uniforme a memoria**): la arquitectura ideal de esta organización se muestra en la figura 5.1. Según este modelo, denominado **PRAM** (***parallel random access memory*, memoria paralela de acceso aleatorio**), cualquier procesador puede acceder a cualquiera de las direcciones de la PRAM en el mismo tiempo (idealmente un ciclo). Esto, evidentemente, no es posible porque habrá peticiones simultáneas procedentes de diferentes procesadores y alguna de ellas tendrá que esperar (tiempo de latencia). Una versión más realista de la organización UMA se muestra en la figura 5.2, en que la memoria se ha dividido en módulos y, además, se ha introducido, entre los procesadores y los módulos de memoria, la red de interconexión. La razón del nombre de este tipo de organización se debe a que todos los módulos de memoria son equivalentes desde el punto de vista de cada procesador.

NUMA (***nonuniform memory access*, acceso no uniforme a memoria**): con esta organización cada procesador posee una memoria local a la que accede con un tiempo de acceso reducido, sin embargo, también puede acceder a los módulos de memoria correspondientes a los demás procesadores, pero debe ser a través de los mismos y con un tiempo de acceso mayor. En la figura 5.3 se muestra esquemáticamente este tipo de organización. Una mejora que suele incorporarse a este tipo de sistemas, es una memoria caché entre cada uno de los procesadores y su memoria local. Esta mejora se muestra esquemáticamente en la figura 5.4. El espacio de direcciones de este tipo de máquinas puede ser local o global, en otras palabras, cada elemento de proceso podrá tener su propio espacio de direcciones o éste será común para todos, estando la memoria distribuida entre todos ellos.

COMA (***caché-only memory architecture*, arquitectura sólo con memoria caché**). Uno de los inconvenientes de la organización NUMA es que el programador tiene que tener mucho cuidado en la **distribución de los datos**, es decir, en situar, dentro de lo posible, en la memoria de cada procesador los datos que mayoritariamente vaya a usar ese procesador. No distribuir los datos correctamente puede hacer que el tiempo de ejecución de un proceso en un sistema con organización NUMA empeore sensiblemente. Con la organi-

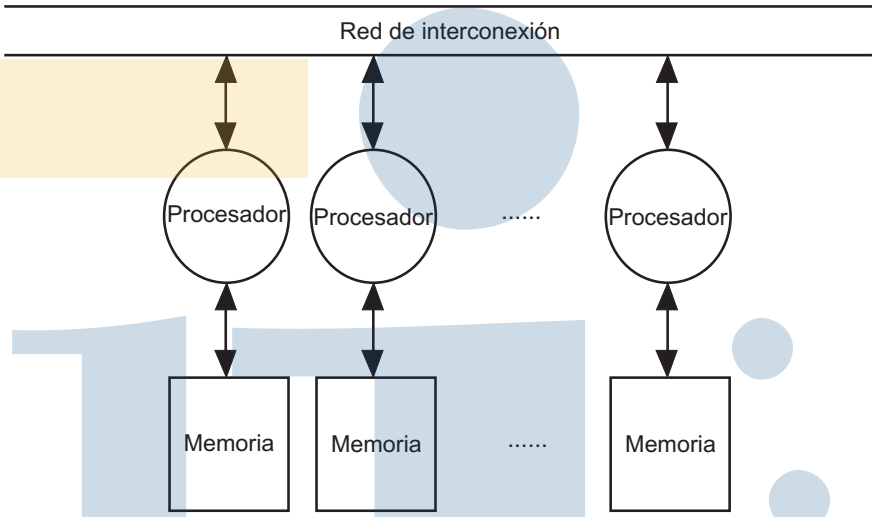

Fig. 5.3. Sistema con acceso a memoria no uniforme (NUMA).

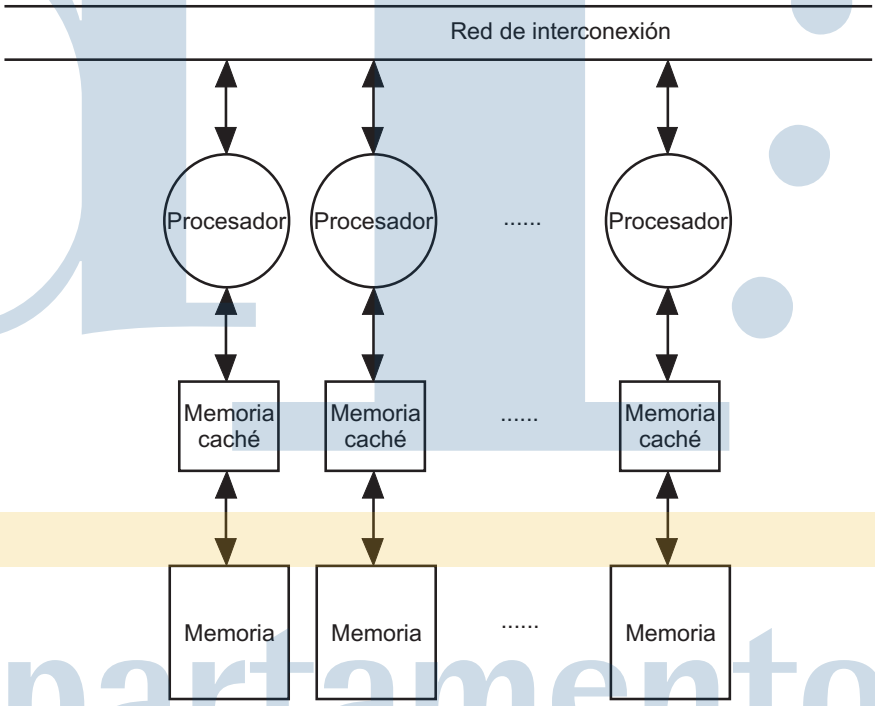

Fig. 5.4. Sistema con acceso a memoria no uniforme (NUMA) y memorias caché locales.

zación COMA, este inconveniente se resuelve, ya que todas las memorias locales de los procesadores se tratan como memorias caché en el sentido de que se van cargando bajo demanda. Ello hace que, en las máquinas con organización COMA, *la distribución de los datos se realiza dinámica y automáticamente*, esto es así porque cada memoria local (llamada **memoria de atracción**) va cargando copias de los bloques que necesita del resto de las memorias locales. Un esquema de este tipo de sistemas se muestra en la figura 5.5.

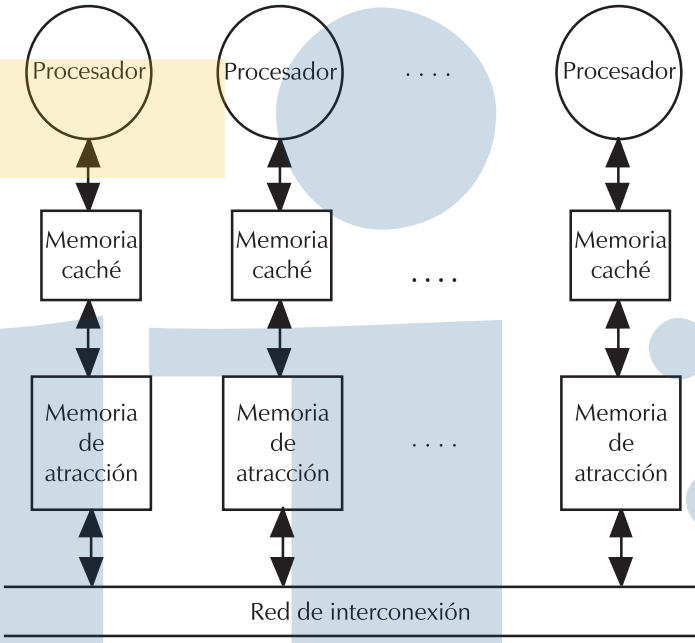

Fig. 5.5. Arquitectura sólo con memoria caché (COMA).

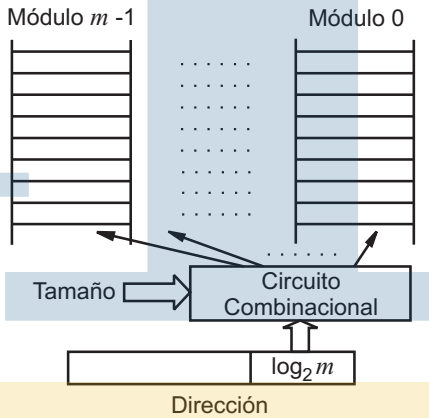

Fig. 5.6. Memoria entrelazada.

5.3. Memorias entrelazadas

La mayoría de los sistemas paralelos emplean memorias entrelazadas, ya que permiten la lectura de varias palabras simultáneas, lo que es muy adecuado para este tipo de sistemas. Como ya debe conocerse, y se muestra en la figura 5.6 (Bastida, 1995), una memoria entrelazada de m vías tiene m módulos independientes que se activan mediante los $\log_2 m$ últimos bits de la dirección. El resto de la dirección llega a todos los módulos. Esto hace que todos los módulos de la memoria puedan trabajar en paralelo, lo que hace a estas memorias muy adecuadas para los multiprocesadores. Si el número de módulos es grande, se necesita algún medio para organizar los accesos si el bus de datos final tiene menos ancho que todos los módulos de memoria juntos.

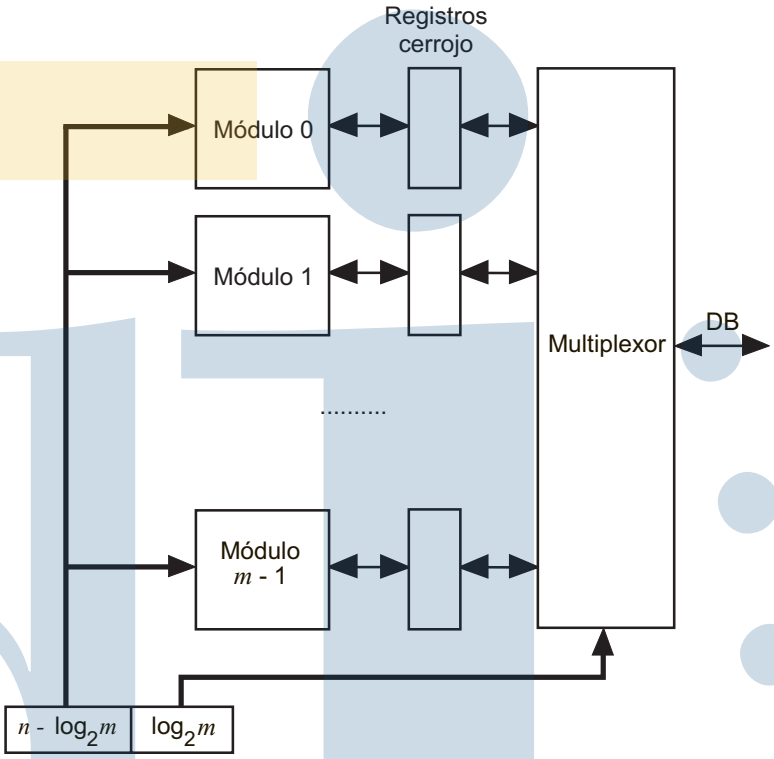

Fig. 5.7. Organización de una memoria entrelazada para acceso S (simultáneo).

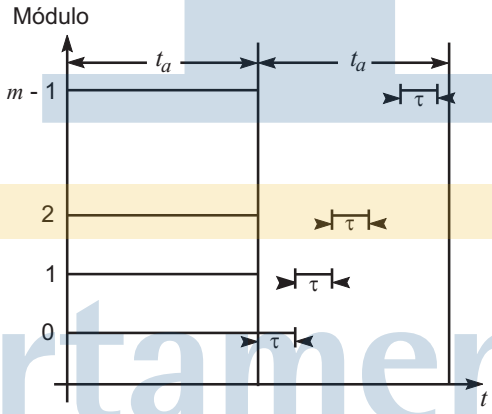

Fig. 5.8. Diagrama temporal del funcionamiento de una memoria entrelazada con acceso S cuando accede a direcciones consecutivas.

Este medio está proporcionado por los métodos de acceso descritos en los apartados siguientes.

5.3.1. Acceso S (simultáneo)

El acceso S (simultáneo), cuyo esquema se muestra en la figura 5.7, se basa en simultanear los accesos en todos los módulos y, además, tratar de superponer ese tiempo con la salida de los datos de memoria. En los accesos a memoria podemos claramente distinguir dos tiempos: el tiempo de acceso dentro de cada módulo (**ciclo mayor**, t_a), y el tiempo de transferencia entre el buffer de memoria y el registro donde se necesita la información (**ciclo menor**, τ). El acceso S basa su funcionamiento en superponer, en la medida de lo posible, ambos ciclos: los m módulos de memoria acceden todos simultáneamente y, mientras se está realizando el siguiente acceso, se aprovecha para ir transfiriendo secuencialmente los datos a través del multiplexor. Se debe procurar que

$$m\tau < t_a$$

porque si no, se produce **rebase de datos** (***overrun***). El rendimiento de la memoria es mejor cuanto más se acerque $m\tau$ a t_a, siempre que no lo sobrepase (esto se puede controlar dando el valor más adecuado a m). Este tipo de memorias funciona muy bien cuando los datos a los que accede son consecutivos (ver figura 5.8). El tiempo total que se necesita para acceder a k palabras consecutivas, comenzando en el módulo i, será:

$$T = \begin{cases} t_a + k\tau, & i + k \leq m \\ 2t_a + (k - m + i)\tau, & i + k > m \quad (k \leq m) \end{cases}$$

La explicación del caso en que $i + k > m$ se basa en que se necesitarán dos lecturas: una para leer las $m - i$ primeras palabras, y otra para leer las restantes, que serán $k - (m - i) = k - m + i$; los ciclos menores de los primeros accesos se superpondrán con el ciclo mayor de las siguientes por lo que su tiempo no se deberá contabilizar.

El ancho de banda máximo estacionario, medido en palabras por segundo, para una memoria con acceso S será:

$$\frac{1}{\frac{t_a}{m}} = \frac{m}{t_a}$$

Claro está que este ancho de banda está calculado para el acceso a palabras contiguas y prescindiendo del tiempo perdido en la primera palabra. En otras condiciones, el ancho de banda disminuirá.

La configuración de acceso S es ideal para acceder a vectores contiguos o para la búsqueda de instrucciones secuenciales. No obstante, este tipo de memoria pierde notablemente su rendimiento si las palabras accedidas no están en direcciones contiguas.

5.3.2. Acceso C (concurrente)

La diferencia entre el acceso C y el acceso S, estudiado en el apartado anterior, radica en que el acceso C no es síncrono, sino que cada módulo puede trabajar por separado, de forma asíncrona, leyendo direcciones diferentes. Cada módulo tiene su propio registro de dirección (incluso podría tener una cola de direcciones) y puede ir trabajando paralelamente a los demás. Evidentemente, el hardware necesario para controlar las peticiones será mucho más complicado que en una memoria con acceso S. El diagrama de tiempos puede ser similar al de una memoria

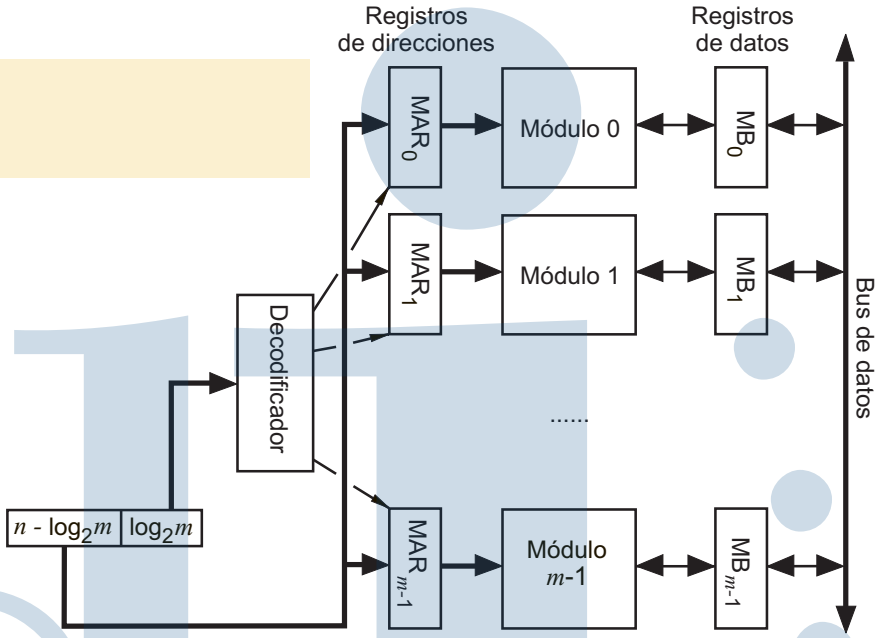

Fig. 5.9. Organización de una memoria entrelazada para acceso C (concurrente).

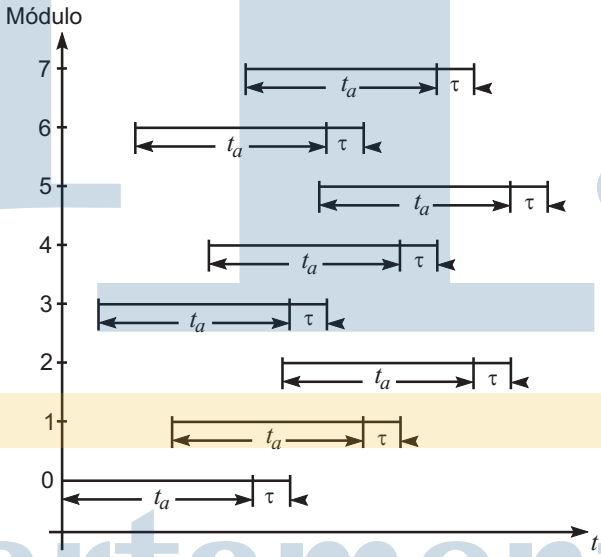

Fig. 5.10. Diagrama temporal del funcionamiento de una memoria entrelazada con acceso C cuando accede a las componentes de un vector con separación 3.

de acceso S con la gran diferencia de que los accesos no tienen que ser contiguos. Sin embargo, existe un problema: puede haber un conflicto en algún módulo en que las peticiones sean demasiado rápidas; por ello, el hardware deberá cuidar también ese aspecto y retrasar las peticiones que no puedan ser atendidas. Las memorias con acceso C funcionan bien, tanto para el acceso a vectores contiguos como para el acceso a vectores cuya separación entre elementos sea primo respecto al número de módulos. Si la separación de los vectores es diferente a ésas, habrá

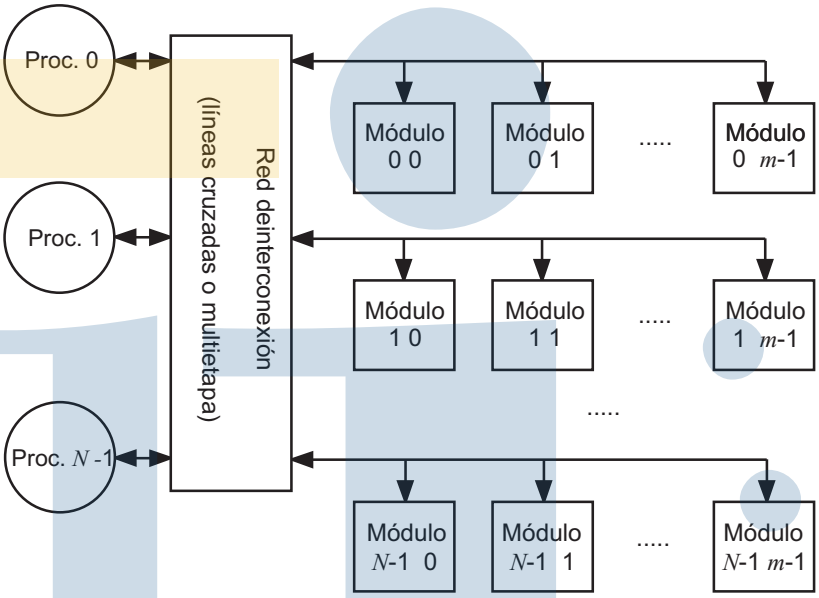

Fig. 5.11. Organización de una memoria entrelazada para acceso C/S.

que efectuar algunas esperas debidas a conflictos en alguno de los módulos. En la figura 5.10 se muestra el diagrama de tiempos del acceso a un vector con separación 3 en una memoria entrelazada de 8 módulos con acceso C.

Como fácilmente puede apreciarse el ancho de banda de una memoria con este tipo de acceso depende mucho de las secuencias de direcciones solicitadas.

En la figura 5.9 se muestra un esquema muy abreviado del principio de funcionamiento de una memoria con acceso C.

5.3.3. Acceso C/S

Es posible, en un mismo sistema paralelo, mezclar los dos tipos de organización analizados en los apartados anteriores: en este caso se habla de acceso C/S. Una de las posibilidades de mantener este tipo de organización se muestra en la figura 5.11 donde, conectados a cada uno de los buses, podría haber m módulos de memoria entrelazados para permitir acceso S y, además, los N buses podrían permitir acceso C ya que cada uno de ellos se conecta a un procesador diferente a través de la red de interconexión. Con acceso C/S el ancho de banda máximo que podría conseguirse, en palabras por segundo, sería:

$$\frac{mN}{t_a}$$

Siendo t_a el tiempo de ciclo mayor.

Este tipo de organización es muy adecuada para multiprocesadores en que cada uno de los elementos de proceso es vectorial. Ello se debe a que un procesador vectorial tratará frecuentemente de acceder a posiciones de memoria consecutivas, por lo que, una vez conectado el

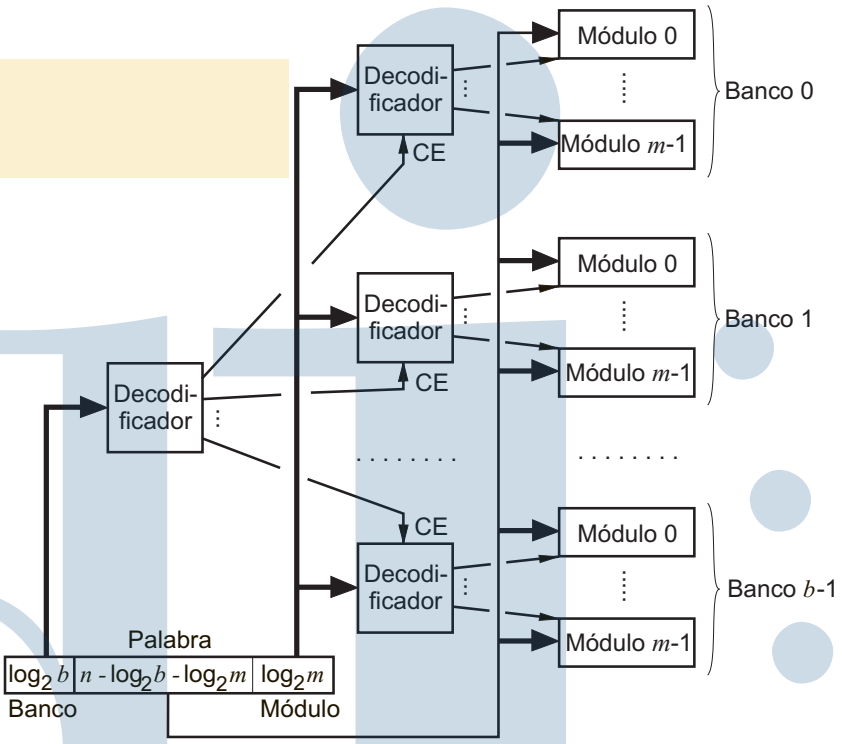

Fig. 5.12. Memoria tolerante a fallos con mb módulos organizados en b bancos

procesador a uno de los buses a través de la red de interconexión, el acceso S a las memorias facilitará la lectura y escritura de los vectores.

5.4. Memorias tolerantes a fallos

Las memorias entrelazadas que hemos visto hasta ahora, tienen la ventaja de que palabras consecutivas se hallan en módulos distintos. Desde el punto de vista de las memorias tolerantes a fallos, esto es un inconveniente, porque, si uno de los módulos falla, no lo podremos aislar del resto de la memoria. Se puede combinar el entrelazado convencional con el llamado **entrelazado de orden superior** en que la memoria se divide en diferentes **bancos** en función del valor de los bits de orden más alto de la dirección. Con esta idea, lo que se pretende es poder aislar un banco completo en caso de fallo. Esto puede conseguirse ahora porque cada banco sí recoge un rango de direcciones consecutivas. Un esquema de una memoria tolerante a fallos con b bancos y mb módulos se muestra en la figura 5.12.

5.5. Coherencia caché

En un multiprocesador, la contención de la memoria puede limitarse si se añade una memoria caché local a cada procesador. Esto, además de las ventajas habituales de la memoria caché,

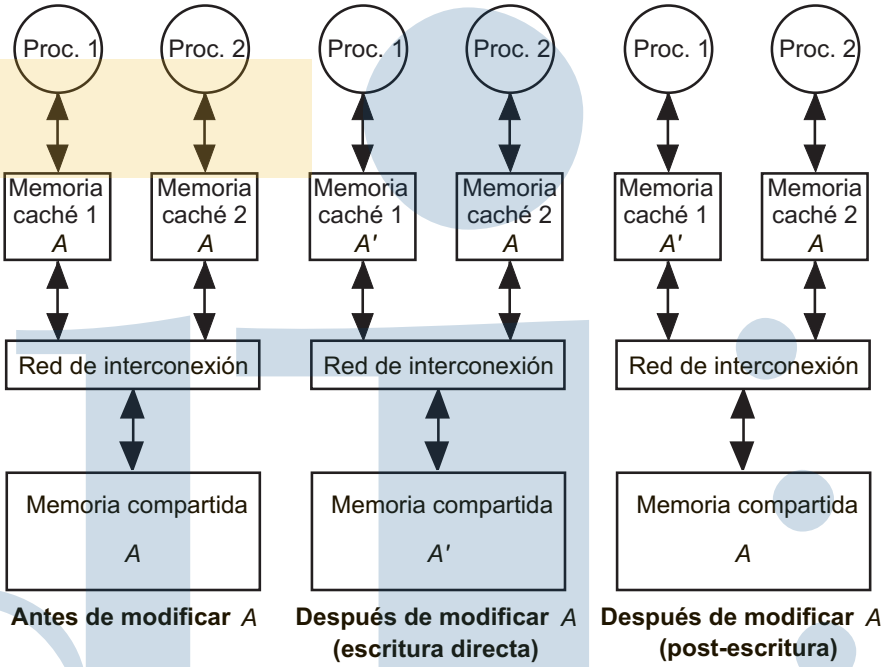

Fig. 5.13. Problema de coherencia caché planteado por una variable compartida.

evitará muchos accesos a la memoria compartida, lo que también liberará recursos en la red de intercomunicación. Sin embargo, la incorporación de estas cachés locales, añade un problema denominado **coherencia caché** que consiste en asegurar que todas las cachés y la memoria compartida contengan el mismo valor de las variables.

Se dice que un sistema de memoria es **coherente** *si el dato devuelto por una instrucción LOAD es* ***siempre*** *el valor dado por la anterior instrucción STORE sobre la misma dirección* (se está suponiendo que la máquina donde se trabaja es una máquina de tipo registro-registro que no es capaz de efectuar operaciones sobre operandos en memoria).

Como es sabido (ver, por ejemplo, (Bastida, 1995)), cuando se utiliza memoria caché en un procesador convencional, existen dos políticas de actualización: el **almacenamiento directo** (***write through***) y la **postescritura** (***write back***): el segundo de estos métodos sería poco conveniente en los multiprocesadores ya que fomentaría la inconsistencia entre cada memoria caché local y la memoria compartida. Para comprender mejor esto obsérvese la figura 5.13: en ella se supone que las memorias cachés locales de los procesadores 1 y 2 tienen sendas copias de la variable A, entonces, el procesador 1 escribe en esa variable A, pasando a valer A'; tanto en el caso de la escritura directa como en el de la post-escritura, existe inconsistencia entre las cachés locales de ambos procesadores, sin embargo, en el caso de la postescritura esta inconsistencia se agrava; ello es debido a que, en este caso y hasta que el bloque afectado se descargue, también habrá inconsistencia entre la memoria caché y la memoria compartida, por ello, si ese bloque se cargara en otra memoria caché local, su valor sería incorrecto. El almacenamiento directo es más apropiado para multiprocesadores ya que modifica simultáneamente cada variable en memoria caché y principal. Este método no es tan lento como podría parecer, ya que la escritura en memoria principal puede simultanearse con el trabajo del procesador; por ello, el almace-

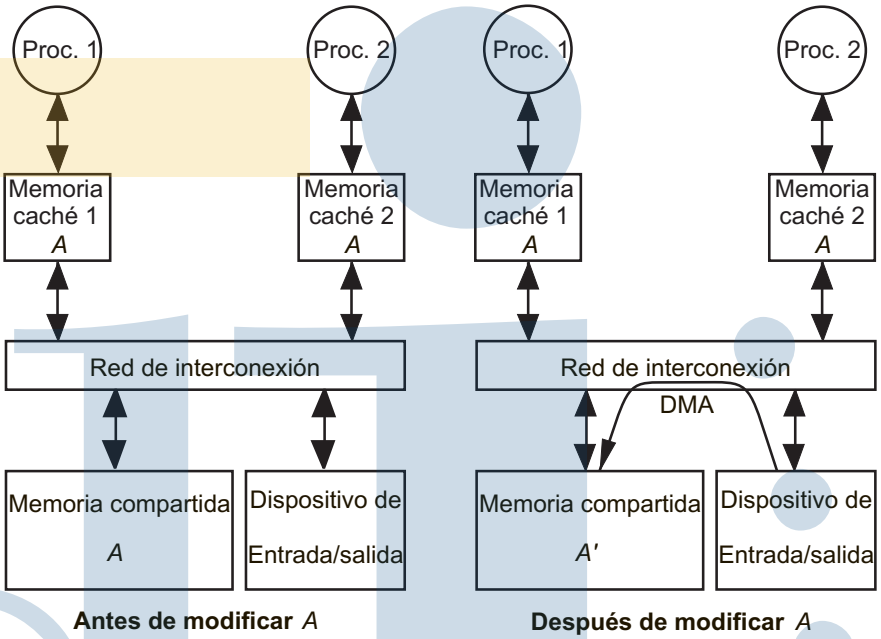

Fig. 5.14. Problema de coherencia planteado por la entrada/salida por DMA.

namiento directo no causará retrasos salvo que existan varias escrituras a memoria demasiado cercanas, y ello no debe ocurrir muy frecuentemente en procesadores de tipo registro-registro. Sin embargo, como se ha visto antes, el almacenamiento directo no garantiza la coherencia entre las diferentes cachés de un sistema multiprocesador. Otra fuente de inconsistencias es el acceso directo a memoria (DMA) por parte de los dispositivos de entrada/salida hacia (o desde) la memoria compartida. En la figura 5.14 se ha supuesto que existen copias de una variable A en las cachés de los procesadores 1 y 2: en ese momento se efectúa una operación de DMA, por parte de algún dispositivo de entrada/salida, que modifica la variable A. En este caso las copias de A en las cachés se vuelven inconsistentes. Una posible solución (figura 5.15) sería que cada dispositivo de E/S (y por ello, cada controlador de DMA existente) esté asociado a alguno de los procesadores, de forma que acceda a la caché de ese procesador.

Existen diferentes tipos de técnicas para evitar o resolver el problema de la coherencia caché, estas técnicas pueden efectuarse por hardware, software, o por una combinación de ambos. Por otra parte, los métodos software pueden ser tanto **estáticos**, es decir que resuelven el problema cuando el programa se compila, como **dinámicos**, que lo resuelven en ejecución; también hay métodos que son una mezcla de ambos. Analizaremos algunas de esas técnicas (Dubois *et al.* , 1988):

1. **Caché compartida**: con esta técnica se evita el problema de la coherencia caché de raíz ya que no se permite que cada procesador tenga una caché privada y sólo podría haber una caché compartida. El inconveniente de este método radica en que la caché está alejada de los elementos de proceso lo que disminuye su efectividad ya que la memoria caché no podrá ser tan rápida (por la capacidad eléctrica de los conductores). Para evitar las inconsistencias debidas a las operaciones E/S por DMA, éstas deben efectuarse a través de la caché compartida.

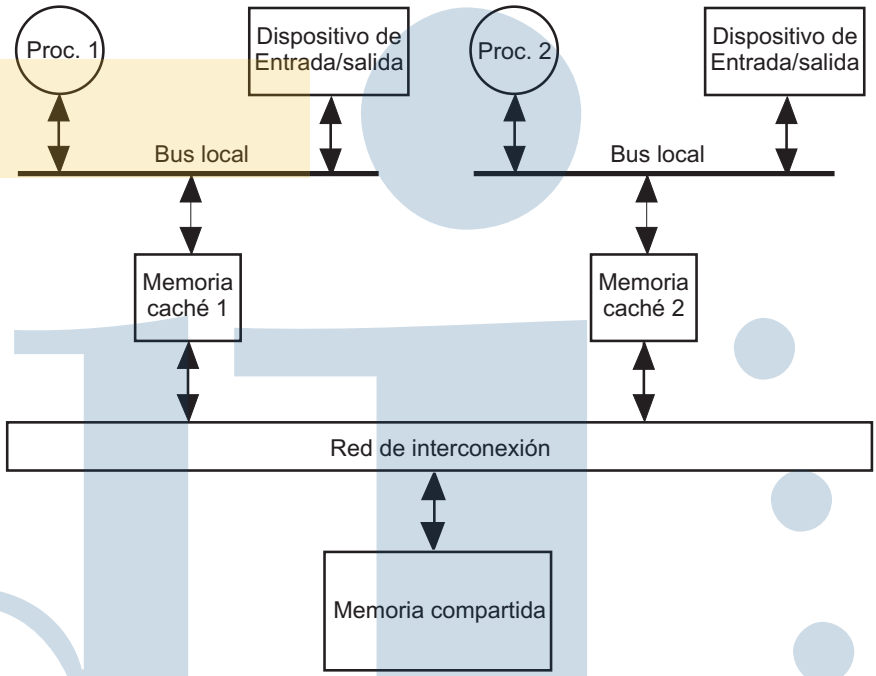

Fig. 5.15. Solución parcial al problema de las inconsistencias debidas al DMA.

2. **Prohibir la entrada en caché a los datos compartidos modificables**: con este método sólo pueden entrar en caché el código, los datos no compartidos o los datos compartidos no modificables. Para ello, el compilador tiene que etiquetar los datos indicando cuáles son susceptibles de entrar en las memorias cachés locales y cuáles no. Este método tiene el inconveniente de que necesita algo de tiempo adicional para analizar las etiquetas puestas por el compilador, lo que degrada un poco el rendimiento. Por otra parte, el método en sí mismo no resuelve las inconsistencias provocadas por el acceso directo a memoria; podría resolverlo si el compilador considera como bloques modificables a los afectados por DMA. Sin embargo, esto sólo será posible si el compilador conoce que direcciones pueden ser modificadas por el DMA y esto no siempre es así debido al uso de apuntadores, longitudes de bloque de DMA que sólo se conozcan en ejecución, etc., por otra parte, para salvaguardar la coherencia en este caso, deben conectarse los controladores de DMA a las cachés locales de los procesadores.

3. **Limpieza de caché** (***cache flush***): Los datos compartidos que puedan ser modificados se protegen mediante una sección crítica. Si un procesador trabaja en una de esas secciones críticas y modifica datos en su memoria caché, cuando acaba dicha sección crítica, invalida esos bloques, si estuvieran cargados en otras cachés, y, por supuesto, aplica el método de escritura directa para actualizar la memoria principal. Esta técnica degrada algo el rendimiento debido a la sobrecarga introducida por la invalidación de las cachés que obliga a enviar mensajes a todos los demás procesadores. En principio, el método no resuelve las inconsistencias provocadas por el DMA porque los controladores de DMA funcionan de forma independiente a los procesadores y su acción no se puede incluir en una sección crítica.

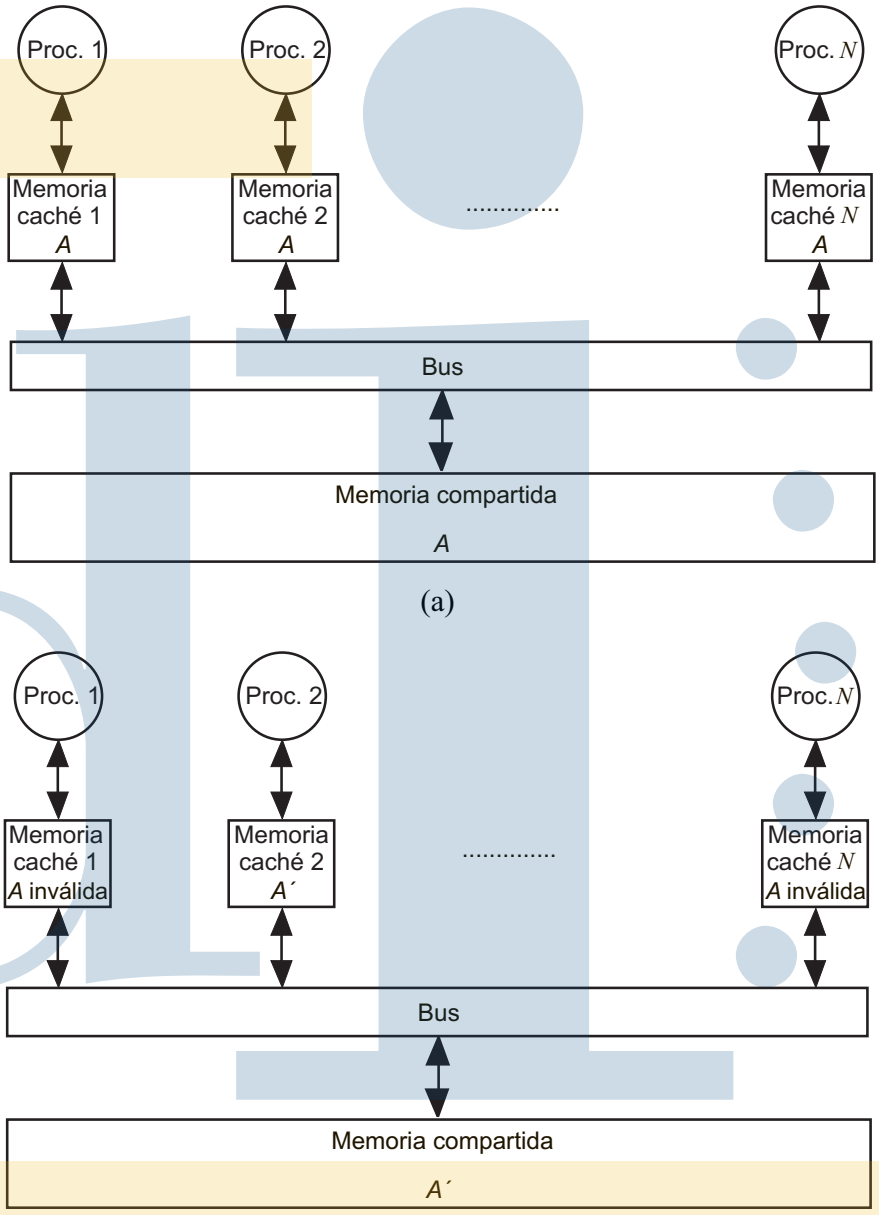

Fig. 5.16. Protocolos inspectores (*snoopy protocols*): (a) Situación inicial, (b) Situación después de una escritura con protocolo de invalidación (*continúa*).

4. **Protocolos inspectores (*snoopy protocols*)**: Si estamos en una estructura de bus (figura 4.13(a)), es posible establecer protocolos para proteger a las memorias caché contra las inconsistencias. Esto es posible porque, en una estructura de bus, todos los dispositivos conectados al bus tienen acceso a las informaciones que por él circulan. Esto hace que, siempre que se emplee almacenamiento directo, todas las memorias cachés conectadas al bus sepan cuándo se está escribiendo en alguno de sus bloques (inspeccionando las líneas del bus). Los protocolos inspectores pueden ser de dos clases:

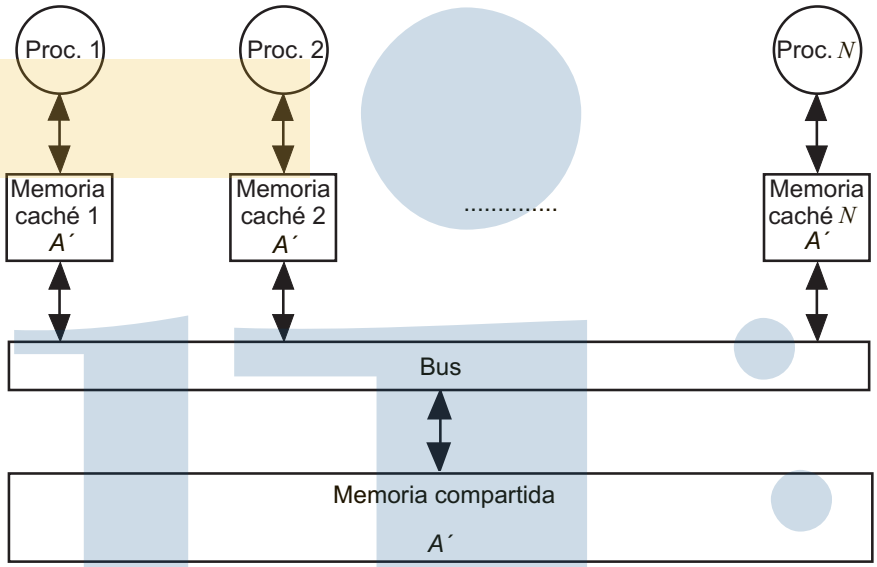

Fig. 5.16. Protocolos inspectores (*snoopy protocols*): (c) Situación después de una escritura con protocolo de actualización (*conclusión*).

Protocolos de invalidación: en estos protocolos se invalidan las copias en caché de los bloques en que se producen escrituras (claro está, exceptuando la correspondiente al procesador que ha efectuado la escritura). Cuando una de las caché detecta, inspeccionando las líneas del bus, que se ha escrito en alguno de sus bloques, invalida esa copia del bloque.

Protocolos de actualización: si se utiliza este tipo de protocolo se deben actualizar todas las copias de los bloques modificados. Cuando una caché detecta que una dirección correspondiente a alguno de sus bloques ha sido modificada, la actualiza en su copia con la información circulante por el bus.

Una muestra de este tipo de protocolos se ilustra en la figura 5.16, en que puede verse una situación inicial con N copias de la variable A en las memorias caché locales de N procesadores. Supongamos, que en esa situación, uno de los procesadores cambia el valor de A en su caché; entonces, si se emplea un protocolo de invalidación (figura 5.16(b)), todos los controladores de caché observarán la escritura de la variable A en su inspección continua del bus: aquéllos que contengan en su caché una copia de la variable A invalidarán ese bloque. Por el contrario, si se emplea un protocolo de actualización, cuando los controladores detecten la escritura de la variable A, actualizarán su contenido con el valor que lean directamente del bus (figura 5.16(c)).

Ambos tipos de protocolo tienen algunos inconvenientes: los protocolos de actualización pierden tiempo haciendo actualizaciones de variables que luego pueden no volver a usarse, debiendo actualizarse otra vez; por otra parte, los protocolos de invalidación tienen también un importante inconveniente conocido como **falsa compartición** (***false sharing***). La falsa compartición es un fenómeno que consiste en que, por compartirse una sola variable de un bloque, el protocolo interpreta que se está compartiendo todo el bloque. Esto causa una disminución en la tasa de aciertos artificial debido a la invalidación

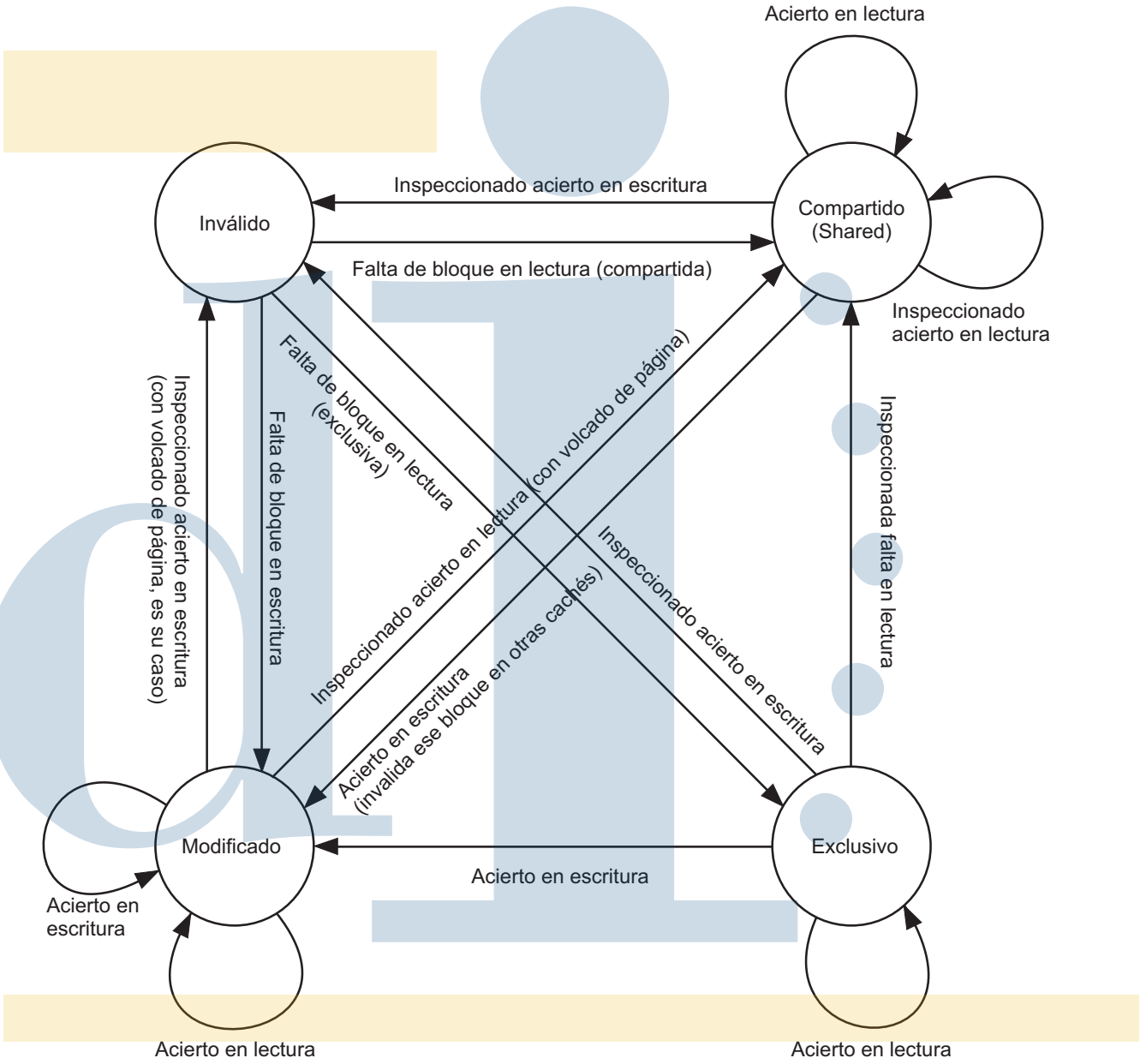

Fig. 5.17. Estados posibles de un bloque bajo el protocolo MESI y las posibles transiciones entre ellos.

de bloques en los que sólo se comparta una variable. Una solución a este problema es dimensionar adecuadamente el tamaño del bloque para que, en caso de invalidación, el número de datos invalidados no sea excesivo. Contrastando ambos métodos, en general, se prefieren los protocolos de invalidación.

5. **Protocolo MESI** o **Illinois** (Papamarcos & Patel, 1984): Este protocolo es, en realidad, una variante de los protocolos inspectores de invalidación. Tiene además la propiedad de que soporta que la memoria caché trabaje con postescritura. Según este protocolo, a cada bloque de memoria caché se le asocian dos bits que se corresponden con uno de estos

cuatro estados, cuyas iniciales dan nombre al protocolo:

Modificado: el bloque de la caché ha sido modificado (por tanto será diferente del original en memoria principal). El bloque sólo es válido en esa caché.

Exclusivo: el bloque es una copia exacta del original en memoria principal y, además, no existen copias de ese bloque en ninguna otra caché.

Compartido (*shared*): el bloque es copia exacta del correspondiente en memoria principal, pero puede haber alguna copia suya en alguna otra caché.

Inválido: los datos de ese bloque no son válidos, esto suele ser debido a que alguna otra copia del bloque ha sido modificada por alguno de los demás procesadores.

La figura 5.17 muestra un diagrama de los estados anteriores y de todas las transiciones posibles entre ellos, se indican también las causas de esas transiciones. Veamos pormenorizadamente algunas de esas transiciones:

Falta de bloque en lectura: cuando ocurre una falta de este tipo, el procesador comienza una lectura de memoria para cargar en caché el bloque solicitado. Pueden darse cuatro situaciones:

- Si algún otro procesador tiene una copia sin modificar de ese bloque en estado exclusivo, el controlador de caché de ese procesador lo advertirá, por la inspección de las líneas del bus, y entonces lo pondrá en estado compartido y enviará una señal para que el bloque de la caché que originalmente ha provocado la falta también quede en estado compartido.
- Si alguno (o varios) procesadores tiene en su caché una copia de ese bloque en estado compartido, envían una señal o código al procesador que ha provocado la falta para poner el bloque cargado en estado compartido.
- Si alguno de los procesadores tiene en su caché una copia modificada del bloque, mandará una señal o código al procesador que ha provocado la falta para que reintente la operación. Mientras tanto, el procesador que tiene la copia modificada, tratará de volcarla a memoria principal y cambiará el estado del bloque de modificado a compartido. Cuando esta transición ocurra, el procesador que produjo la falta podrá ya hacer una copia del bloque en su caché, dejándola también en estado compartido.
- Si ninguna otra caché tiene copia del citado bloque, entonces quedará en estado exclusivo.

Falta de bloque en escritura: en esta situación, y si se ubica en escritura, se inicia la lectura del bloque solicitado para cargarlo en caché. Esto activará en el bus una línea, denominada RWITM (*read with intent to modify*). Cuando el bloque se cargue, se marcará inmediatamente como modificado, a este respecto debemos tener en cuenta a las restantes cachés, por lo que pueden ocurrir dos casos:

- Alguna de las demás cachés puede tener una copia modificada de este bloque, en este caso, deberemos esperar a que esa caché vuelque su copia a memoria principal, para luego, marcarla como inválida. Cuando la copia haya sido volcada e invalidada, ya se puede proceder a la carga del bloque solicitado.

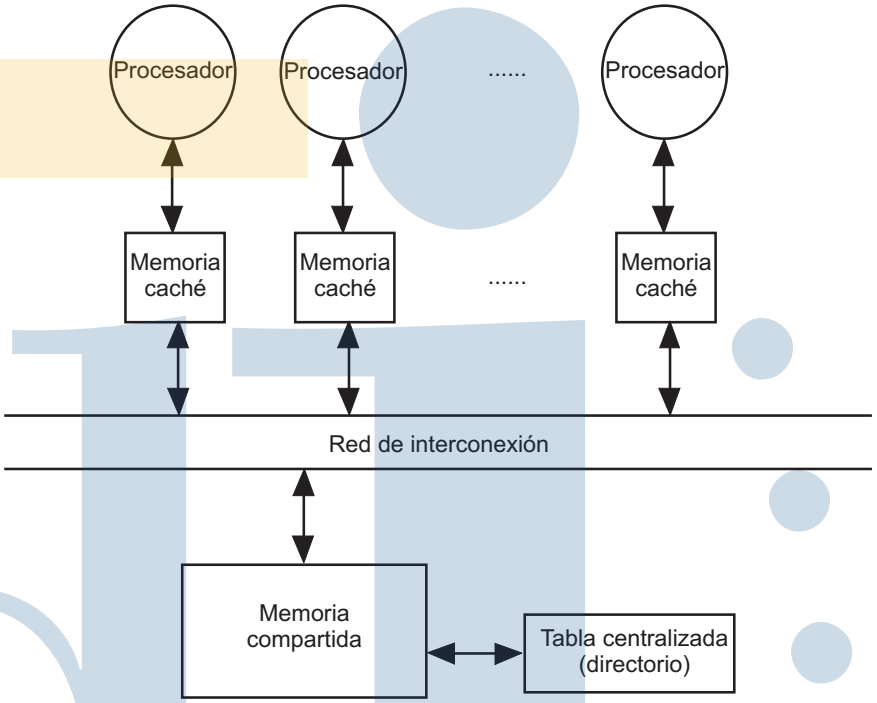

Fig. 5.18. Esquema de funcionamiento de un protocolo de directorio.

- Las demás cachés no tienen ninguna copia modificada del bloque, aunque puedan tenerla en otro estado. En este caso no hay inconveniente para cargar el bloque solicitado, pero las demás copias deben ser invalidadas.

Acierto en escritura: cuando ocurre una escritura en un bloque que ya está en caché, ese bloque puede estar en varios estados:

- **Compartido**: antes de actualizar el bloque, el procesador tiene que conseguir acceso exclusivo para el mismo. Para obtenerlo, mandará el intento de escritura hacia el bus (RWITM) para que todos los restantes procesadores, que disponen de copias del bloque, las invaliden (y lo harán al inspeccionar las líneas del bus y advertir que se ha producido una escritura en un bloque del que tienen una copia). Una vez conseguido el acceso exclusivo y efectuada la escritura el bloque pasará a estado modificado.
- **Exclusivo**: en este caso la operación se limita a actualizar el bloque afectado, y la memoria principal, si la política de actualización es la escritura directa, si no lo fuera, cambia el bloque a estado modificado.
- **Modificado**: se opera de forma similar al caso anterior, ya que, al estar el bloque modificado, el procesador ya tiene control exclusivo sobre él.

6. **Tabla centralizada**. Estos métodos también se denominan **protocolos de directorio** (Censier & Feautrier, 1978): Los protocolos inspectores descritos anteriormente tienen el inconveniente de que, en principio, sólo sirven para redes de interconexión de bus, en que todos los elementos conectados al mismo pueden escuchar lo que por él circula. Si

se trata de aplicar este tipo de protocolos en sistemas con otras redes de interconexión, será necesario que cada vez que se efectúen escrituras en memoria se envíen mensajes de difusión con el fin de que todos los elementos de proceso con caché puedan invalidar o actualizar sus datos. Estos mensajes aumentarían notablemente el tráfico a través de la red de interconexión aumentando la latencia de la misma. Ello hace necesario otro tipo de protocolos para garantizar la coherencia. Una alternativa a los protocolos inspectores son los protocolos de directorio, en que el sistema lleva una tabla global, que se localiza en la memoria compartida (ver figura 5.18), y que indica los bloques almacenados en cada caché y el tipo de acceso a cada bloque (sólo lectura, lectura/escritura). Únicamente una de las cachés puede tener una copia de un bloque para lectura/escritura. Por tanto, antes de que un procesador pueda escribir en su copia local de un bloque, tiene que pedir acceso exclusivo a ese bloque. Antes de conceder el acceso, el controlador de la memoria central mandará un mensaje a todos los procesadores para que invaliden sus copias de ese bloque (si es que las tienen). Cuando el controlador recibe la confirmación de que los mensajes han llegado, y que, por tanto, las copias restantes han sido invalidadas, concede el acceso al procesador que ha solicitado la escritura, asegurando así el acceso exclusivo. Esta técnica es eficaz ante las inconsistencias producidas por el DMA. El problema de este método es que los accesos continuos a la tabla centralizada pueden causar cuellos de botella en la zona de memoria donde se encuentre; por otra parte, las consultas a la tabla causan una sobrecarga en los procesadores que degrada el rendimiento.

Los protocolos de directorio muchas veces se utilizan como refuerzo de otras técnicas, para asegurar su buen funcionamiento.

Bibliografía y referencias

BASTIDA, J. 1995. *Introducción a la Arquitectura de Computadores*. Universidad de Valladolid.

CENSIER, L., & FEAUTRIER, P. 1978. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, **27**(12).

DUBOIS, M., SCHEURICH, C., & BRIGGS, F.A. 1988. Synchronization, Coherence, and Event Ordering in Multiprocessors. *IEEE Computer*, **21**(2).

HWANG, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill.

LENOSKI, D.E., & WEBER, W-D. 1995. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers.

ORTEGA, J. ET AL. 2005. *Arquitectura de Computadores*. Thomson.

PAPAMARCOS, M., & PATEL, J. 1984 (May). A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *In: Proceedings of the 11 International Symposium on Computer Architecture*.

STALLINGS, W. 2005. *Computer Organization and Arquitecture: Designing for Performance*. 7 edn. Prentice-Hall. Existe traducción al castellano: Organización y arquitectura de computadores, 7ª edición, Prentice-Hall, 2006.

CUESTIONES Y PROBLEMAS

5.1 Establecer las cotas superior e inferior del ancho de banda de una memoria con acceso C.

5.2 Sea una memoria entrelazada con 32 vías que se conecta para proporcionar acceso S. El ciclo mayor de esta memoria dura 1 μsg. y el menor, 30 nsg.:

a) Calcular el tiempo de acceso total para un dato de una palabra.

b) Calcular el tiempo de acceso para un vector con separación unitaria de 15 palabras que comience en la dirección 04A0F0H.

c) ¿Variaría en algo la respuesta al apartado anterior si el vector comenzara en la dirección A5ABFDH?

d) ¿Cuánto tiempo se tardará en acceder a un vector de 40 palabras contiguas que comience en dirección 0BA420H?

5.3 Sea una memoria entrelazada con 64 vías que se conecta para proporcionar acceso C. El ciclo mayor de esta memoria dura 2 μsg. y el menor, 30 nsg.:

a) Calcular el tiempo de acceso total para un dato de una palabra.

b) Calcular el tiempo de acceso para un vector con separación unitaria de 15 palabras que comience en la dirección 04BA30H.

c) ¿Variaría en algo la respuesta al apartado anterior si el vector comenzara en la dirección 054BFFH?

d) Calcular el tiempo de acceso total para un vector de 128 elementos que comience en la dirección 09FEA0H y que tenga una separación entre elementos de 2 palabras.

e) Supongamos que, a partir de cierto momento, se solicitan la siguiente secuencia de direcciones de memoria (todas en hexadecimal):

067FA1, 04AB02, 050A81, 067A42, 034782, 065022

Calcular el tiempo total de acceso a todas esas palabras.

5.4 Sea una memoria entrelazada con 4 vías en que se está evaluando las posibilidades de conectarla para acceso C o acceso S. Para esa memoria, el tiempo de ciclo menor es t y el de ciclo mayor es $5t$. Calcular los tiempos necesarios, tanto para acceso S como para acceso C, para las siguientes lecturas:

a) 4 palabras de memoria consecutivas.

b) 4 palabras de memoria con separación 3.

c) 4 palabras de memoria con separación 4.

5.5 Considerar los siguientes diseños para una memoria entrelazada y tolerante a fallos con 16 módulos de 2Mg. cada uno:

1. Memoria entrelazada con 16 vías en un solo banco.
2. Memoria entrelazada con 8 vías en 2 bancos.

3. Memoria entrelazada con 4 vías en 4 bancos.

a) Dibujar un pequeño esquema de cada uno de los casos.

b) Para cada caso, especificar la longitud de los campos de una dirección de memoria.

c) Describir las ventajas e inconvenientes de cada diseño.

5.6 Sea una memoria entrelazada tolerante a fallos. Esta memoria dispone de 32 módulos de 64 Mg. organizados en 4 bancos.

a) Indicar el tamaño de cada uno de los campos de la dirección de memoria.

b) Dibujar un esquema simplificado de esa memoria.

c) ¿Cuál será la secuencia de módulos accedidos para leer un vector de 64 elementos con separación unitaria a partir de la dirección 20A9A004H?

d) Calcular el tiempo de acceso a todo el vector del apartado anterior suponiendo que el ciclo mayor consume 0,5 μsg. y el menor, 60 nsg. (supóngase que la memoria funciona con acceso S)

e) Repetir el apartado anterior pero suponiendo que la separación del vector es 3 y que la memoria tiene acceso C.

f) ¿Habría alguna diferencia en el resultado anterior si la memoria tuviera acceso S?

g) Supongamos ahora que se daña el módulo que contiene a la dirección 0B04890AH. ¿Cuánta memoria quedará disponible cuando se detecte la avería?

5.7 Sea un sistema biprocesador en que cada uno de sus procesadores dispone de una memoria caché. Supongamos que en cada uno de esos procesadores se ejecuta una copia del siguiente fragmento de programa:

```
...
A=C+2
B=A+C
B=C+1
...
```

Suponiendo que en cierta ejecución, el programa se ejecuta primero en el procesador 1 y luego, con una sola instrucción de diferencia en el procesador 2:

a) Estudiar bajo qué políticas de extracción y actualización podremos tener problemas de coherencia.

b) Suponiendo que se ubican los bloques para escritura, aunque se utilice escritura directa, explicar como actuaría un protocolo inspector de actualización.

c) Repetir el apartado anterior utilizando un protocolo inspector de invalidación.

5.8 Sea un sistema biprocesador en que cada procesador dispone de una memoria caché con ubicación en escritura y postescritura. En el procesador 1 se ejecuta el código siguiente:

```
t1  LOAD   A,   R1
t2  LOAD   B,   R2
t4  ADD    R1,  R2
t6  STORE  R2,  B
```

mientras que en el procesador 2, el código que se ejecuta es:

```
t3  LOAD   B,   R2
t5  LOAD   C,   R1
t7  ADD    R1,  R2
t8  STORE  R2,  B
```

En ambos códigos, t_i representa el momento de la ejecución de cada instrucción (evidentemente, subíndices mayores representan momentos posteriores). Se puede suponer que las variables residen en bloques diferentes de memoria entre los que no hay conflictos.

a) Si se emplea el protocolo MESI, explicar por qué estados pasará cada uno de los bloques de memoria caché afectados en ambos procesadores.

b) Suponiendo que en la variable `B` se quiera acumular valores anteriores con `A` y `C`, describir alguna técnica que asegure un resultado correcto.