Capítulo 5

Multiprocesadores

Los multiprocesadores son sistemas MIMD basados en varios procesadores funcionando de forma paralela e independiente. La principal característica de los multiprocesadores es que la memoria está compartida, es decir, todos los procesadores comparten el mismo espacio de direccionamiento.

El libro [CSG99] contiene la mayor parte de los temas tratados en este capítulo, además se trata de un libro muy completo sobre multiprocesadores.

5.1. Coherencia de las cachés

La bibliografía sobre la coherencia de caché para multiprocesadores es bastante amplia, casi cualquier libro de arquitectura de computadores incluye alguna cosa. Los más completos sin embargo son [Hwa93], [HP96], [Zar96] y [Fly95]. Los apuntes se han realizado sobre todo a partir del [Hwa93] y del [Zar96]. Del [Fly95] se han extraído algunos de los diagramas de estados y constituye una buena lectura adicional. Del [HP96] se han extraído algunas ideas y definiciones y vienen algunos experimentos sobre rendimiento que no se han incluido en los apuntes.

La memoria es uno de los componentes principales de cualquier sistema. La estructura de la memoria va a tener un importante impacto en el rendimiento general del sistema. Se analizan en esta parte los diferentes protocolos de coherencia fundamentales en los actuales sistemas multiprocesadores con memoria compartida.

En cursos anteriores el alumno se ha encontrado ya con el tema de las cachés y su coherencia en sistemas con un único procesador y una sola caché. No se explicarán aquí los diversos tipos de caché ni su realización, ya que se supone que es un tema conocido. Lo que se expone a continuación es el problema de mantener la coherencia del sistema de memoria en sistemas multiprocesadores con varias cachés y memoria compartida.

5.1.1. El problema de la coherencia de las cachés

Pensemos por un momento cuál es el modelo intuitivo que tenemos de lo que debe ser una memoria. La memoria debe proporcionar un conjunto de direcciones para almacenar valores, y cuando se lea una de estas direcciones debe devolver el último valor escrito

en ella. Es en esta propiedad fundamental de las memorias en la que descansan los programas secuenciales cuando usamos la memoria para comunicar un valor desde un punto del programa donde se calcula a otros puntos donde es usado. También nos basamos en esta propiedad cuando el sistema usa un espacio de direcciones compartido para comunicar datos entre hebras o procesos que se están ejecutando en un procesador. Una lectura devuelve el último valor escrito en esa dirección, sin importar el proceso que escribió dicho valor. Las cachés (antememorias) no interfieren con el uso de múltiples procesos en un procesador, ya que todos ellos ven la memoria a través de la misma jerarquía de cachés. En el caso de usar varios procesadores, nos gustaría poder basarnos en la misma propiedad cuando dos procesos se ejecuten sobre diferentes procesadores de tal forma que el resultado de ejecutar un programa que usa varios procesos sea el mismo independientemente de si los procesos se ejecutan o no en diferentes procesadores físicos. Sin embargo, cuando dos procesos ven la memoria compartida a través de diferentes cachés, existe el peligro de que uno vea el nuevo valor en su caché mientras que el otro todavía vea el antiguo.

El problema de la coherencia en sistemas multiprocesadores se puede ver claramente mediante un ejemplo. Supongamos dos procesadores, cada uno con su caché, conectados a una memoria compartida. Supongamos que ambos procesadores acceden a una misma posición X en memoria principal. La figura 5.1 muestra los contenidos de las cachés y memoria principal asumiendo que ninguna caché contiene inicialmente la variable que vale inicialmente 1. También se asume una caché *write-through*. Después de que el valor en X ha sido escrito por A, la caché de A y la memoria contienen el nuevo valor, pero la cache de B no, y si B lee el valor de X, ¡leerá 1 y no 0 que es el valor de la memoria! Este es el problema de la coherencia de cachés en multiprocesadores.

Time	Acción	Caché A	Caché B	Memoria (X)
1	CPU A lee X	1		1
2	CPU B lee X	1	1	1
3	CPU A escribe 0 en X	0	1	0

Figura 5.1: El problema de la coherencia de cachés con dos procesadores.

Informalmente se puede decir que un sistema de memoria es *coherente* si cualquier lectura de un dato devuelve el valor más recientemente escrito de ese dato. Esta definición, aunque intuitivamente correcta, es vaga y simple; la realidad es bastante más compleja. Esta definición simple contiene dos aspectos diferentes del comportamiento del sistema de memoria, siendo los dos críticos a la hora de escribir programas en memoria compartida. El primer aspecto, llamado *coherencia* definen los datos devueltos por una lectura. El segundo aspecto, llamado *consistencia*, determina cuando un valor escrito será devuelto por una lectura.

Demos primero una definición formal para la coherencia. Se dice que un sistema de memoria es coherente si se cumple:

1. Una lectura por un procesador P de una posición X, que sigue de una escritura de P a X, sin que ningún otro procesador haya escrito nada en X entre la escritura y la lectura de P, siempre devuelve el valor escrito por P.
2. Una lectura por un procesador de la posición X, que sigue una escritura por otro procesador a X, devuelve el valor escrito si la lectura y escritura están suficientemente separados y no hay otras escrituras sobre X entre los dos accesos.

3. Las escrituras a la misma posición están serializadas, es decir, dos escrituras a la misma posición por cualquiera dos procesadores se ven en el mismo orden por todos los procesadores. Por ejemplo, si se escriben los valores 1 y 2 en una posición, los procesadores nunca pueden leer el valor 2 y luego el 1.

La primera condición preserva el orden del programa y debe ser así incluso para monoprocesadores. La segunda condición define lo que se entiende por tener una visión *coherente* de la memoria. Con estas condiciones el problema para tener un sistema coherente queda resuelto, sin embargo, todavía queda el problema de la consistencia.

Para ver que el problema de la consistencia es en realidad complejo, basta con observar que en realidad no es necesario que una lectura sobre X recoja el valor escrito de X por otro procesador. Si por ejemplo, una escritura sobre X precede una lectura sobre X por otro procesador en un tiempo muy pequeño, puede ser imposible asegurar que la lectura devuelva el valor tan recientemente escrito, ya que el valor a escribir puede incluso no haber abandonado el procesador que escribe. La cuestión sobre cuándo exactamente un valor escrito debe ser visto por una lectura se define por el *modelo de consistenciade la memoria.*

Otro ejemplo que ilustra el problema de incoherencia de la memoria se puede ver en la figura 5.2. En esta figura se muestran tres procesadores con cachés propias y conectados a través de un bus común a una memoria compartida. La secuencia de accesos a la localización u hecha por cada procesador es la que se indica mediante los números que aparecen al lado de cada arco. Veamos cuáles son los valores leídos por P_1 y P_2 dependiendo del tipo de caché utilizada.

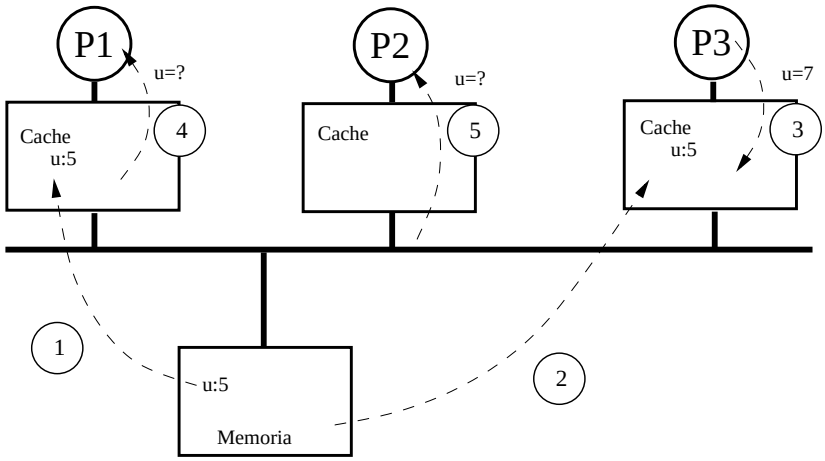


Figura 5.2: Ejemplo del problema de coherencia de las cachés.

- Si consideramos una caché *write-through* la modificación realizada por el procesador P_3 hará que el valor de u en la memoria principal sea 7. Sin embargo, el procesador P_1 leerá el valor de u de su caché en vez de leer el valor correcto de la memoria principal.
- Si considera una caché *writeback* la situación empeora. En este caso el procesador P_3 únicamente activará el bit de modificado en el bloque de la caché en donde tiene almacenado u y no actualizará la memoria principal. Únicamente cuando ese bloque de la caché sea reemplazado su contenido se volcará a la memoria principal. Ya no será únicamente P_1 el que leerá el valor antiguo, sino que cuando P_2 intente leer u y se produzca un fallo en la caché, también leerá el valor antiguo de la memoria principal.

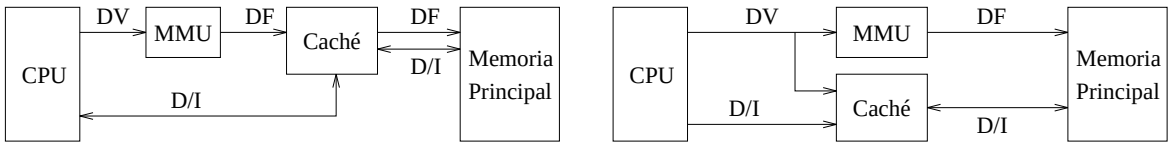
Por último, si varios procesadores escriben distintos valores en la localización u con este tipo de cachés, el valor final en la memoria principal dependerá del orden en que los bloques de caché que contienen u se reemplazan, y no tendrá nada que ver con el orden en que las escrituras a u ocurrieron.

Los problemas de coherencia en la caché también ocurren cuando utilizamos un único procesador en el caso de las operaciones de E/S. La mayoría de estas operaciones se realizan a través de dispositivos DMA con lo que es posible que que el contenido de la memoria principal y la memoria caché dejen de ser coherentes. Sin embargo, dado que las operaciones de E/S son mucho menos frecuentes que las operaciones de acceso a memoria, se han adoptado soluciones sencillas como usar direcciones de memoria que se marcan como no almacenables en la memoria caché o eliminar todos los bloques existentes en las cachés de las páginas que se van a utilizar en la operación de E/S antes de proceder a realizar dicha operación. En la actualidad, casi todos los microprocesadores proporcionan mecanismos para soportar la coherencia de cachés.

5.1.2. Direcciones físicas y virtuales, problema del *aliasing*

Hay dos formas de conectar la caché al procesador. Una manera consiste en colocar la caché después de los bloques TLB (*Transaction Lookaside Buffer*) o MMU (*Memory Management Unit*) que realizan la transformación de dirección virtual a dirección física. A estas cachés conectadas de esta forma se les llama *cachés con direcciones físicas,* y un esquema de su conexionado se muestra en la figura 5.3(a).

La otra posibilidad consiste en conectar la caché directamente a las direcciones del procesador, es decir, a las direcciones virtuales tal y como se muestra en la figura 5.3(b). En este caso tenemos las *cachés con direcciones virtuales.* Ambas tienen sus ventajas e inconvenientes como veremos a continuación.



(a) Caché con direcciones físicas.

(b) Caché con direcciones virtuales.

Figura 5.3: Direcciones físicas y virtuales en la caché. (DV=Dirección Virtual, DF=Dirección Física, D/I=Datos o Instrucciones).

Las cachés con dirección física son sencillas de realizar puesto que la dirección que se especifica en la etiqueta de la caché (*tag*) es única y no existe el problema del *aliasing* que tienen las cachés con dirección virtual. Como no tiene problemas de aliasing no es necesario vaciar (*flush*) la caché, y además el sistema operativo tiene menos *bugs* de caché en su núcleo.

El problema de las cachés con direcciones físicas es que cuesta más tiempo acceder a la caché puesto que hay que esperar a que la unidad MMU/TLB acabe de traducir la dirección. La integración de la MMU/TLB y caché en el mismo VLSI chip, muchas veces incluso con el procesador, alivia este problema.

Lo cierto es que la mayoría de sistemas convencionales utilizan la caché con direcciones físicas puesto que son muy simples y requieren una casi nula intervención del núcleo del sistema operativo.

En la caché con direcciones virtuales, el acceso a la caché se realiza al mismo tiempo y en paralelo con la traducción de dirección por parte de la MMU. Esto permite un acceso a la caché más rápido que con las direcciones físicas.

Pero con las direcciones virtuales surge el problema del **aliasing**. Es a veces frecuente que dos procesos diferentes utilicen las mismas direcciones virtuales cuando en realidad las direcciones físicas van a ser diferentes puesto que nada tiene que ver un proceso con el otro. Cuando esto ocurre, resulta que el índice/etiqueta en la caché coincide en ambos casos cuando en realidad se trata de direcciones reales diferentes. La forma de solucionar esto consiste en vaciar completamente la caché cada vez que hay un problema de *aliasing*, lo cual ocurre siempre que se cambia de contexto en Unix por ejemplo.

Estas operaciones de vaciado continuo de la caché produce un rendimiento muy bajo en la caché y del sistema en general al ocupar el bus en vaciar una y otra vez la caché. En Unix incluso el vaciado de la caché puede no ser suficiente para realizar ciertas operaciones del núcleo, llamadas al sistema y operaciones de I/O. También el depurado de programas resulta prácticamente imposible con este tipo de caché.

Una solución consiste en añadir información sobre el proceso, o contexto, a la etiqueta de la caché, o también, añadir la dirección física, pero estos métodos pueden eventualmente también degradar el rendimiento del sistema.

5.1.3. Soluciones a los problemas de coherencia

Antes de estudiar las diferentes técnicas para solucionar el problema de coherencia, sería interesante realizar una definición más formal de la propiedad de coherencia.

En primer lugar empezaremos con la definición de algunos términos en el contexto de los sistemas de memoria uniprocesador, para después extender dicha definición para multiprocesadores.

Operación de Memoria Un acceso a una dirección de la memoria para realizar una lectura, escritura u operación atómica de lectura-modificación-escritura. Las instrucciones que realizan múltiples lecturas y escrituras, tales como las que aparecen en muchos de los conjuntos de instrucciones complejos, pueden ser vistas como un conjunto de operaciones de memoria, y el orden en que dichas operaciones deben ejecutarse se especifica en la instrucción.

Ejecución atómica Las operaciones de memoria de una instrucción se ejecutan de forma atómica una respecto a la otra según el orden especificado. Es decir, todos los aspectos de una operación deben parecer que se han ejecutado antes de cualquier aspecto de la operación siguiente.

Emisión Una operación de memoria es emitida cuando deja el entorno interno del procesador y se presenta en el sistema de memoria, que incluye las cachés, los buffers de escritura, el bus, y los módulos de memoria. Un punto muy importante es que el procesador sólo observa el estado del sistema de memoria mediante la emisión de operaciones de memoria; es decir, nuestra noción de lo que significa que una operación de memoria haya sido *realizada* es que parezca que ha tenido lugar desde la perspectiva del procesador.

Realizada con respecto a un procesador Una operación de escritura se dice que ha sido realizada con respecto al procesador cuando una lectura posterior por el procesador devuelve el valor producido por esa escritura o una posterior. Una operación de lectura se dice que ha sido realizada con respecto al procesador cuando escrituras posteriores emitidas por el procesador no pueden afectar al valor devuelto por la lectura.

Las mismas operaciones realizadas con respecto a un procesador se pueden aplicar al caso paralelo; basta con reemplazar en la definición "el procesador" por "un procesador". El problema que aparece con el concepto de orden y de las nociones intuitivas de 'posterior' y 'última', es que ahora ya no tenemos un orden definido por el programa sino órdenes de programas separados para cada proceso y esos órdenes de programa interactúan cuando acceden al sistema de memoria. Una manera agudizar nuestra noción intuitiva de sistema de memoria coherente es determinar qué pasaría en el caso de que no existieran cachés. Cada escritura y lectura a una localización de memoria accedería a la memoria física principal y se realizaría con respecto a todos los procesadores en dicho punto, así la memoria impondría un orden secuencial entre todas las operaciones de lectura y escritura realizadas en dicha localización. Además, las lecturas y escrituras a una localización desde cualquier procesador individual deberán mantener el orden del programa dentro de esta ordenación secuencial. No hay razón para creer que el sistema de memoria deba intercalar los accesos independientes desde distintos procesadores en una forma predeterminada, así cualquier intercalado que preserve la ordenación de cada programa individual es razonable. Podemos asumir algunos hechos básicos; las operaciones de cada procesador será realizadas en algún momento dado. Además, nuestra noción intuitiva de "último" puede ser vista como el más reciente en alguno de las hipotéticas ordenaciones secuenciales que mantienen las propiedades discutidas en este párrafo.

Dado que este orden secuencial debe ser consistente, es importante que todos los procesadores vean las escrituras a una localización en el mismo orden.

Por supuesto, no es necesario construir una ordenación total en cualquier punto de la máquina mientras que se ejecuta el programa. Particularmente, en un sistema con cachés no queremos que la memoria principal vea todas las operaciones de memoria, y queremos evitar la serialización siempre que sea posible. Sólo necesitamos estar seguros de que el programa se comporta como si se estuviese forzando un determinado orden secuencial.

Más formalmente, decimos que un sistema de memoria multiprocesador es *coherente* si el resultado de cualquier ejecución de un programa es tal que, para cada localización, es posible construir una hipotética ordenación secuencial de todas las operaciones realizadas sobre dicha localización (es decir, ordenar totalmente todas las lecturas/escrituras emitidas por todos los procesadores) que sea consistente con los resultados de la ejecución y en el cual:

1. las operaciones emitidas por un procesador particular ocurre en la secuencia arriba indicada en el orden en las cuales se emiten al sistema de memoria por ese procesador, y
2. el valor devuelto por cada operación de lectura es el valor escrito por la última escritura en esa localización en la secuencia arriba indicada.

Está implícita en la definición de coherencia la propiedad de que todas las escrituras a una localización (desde el mismo o diferente procesador) son vistas en el mismo

orden por todos los procesadores. A esta propiedad se le denomina *serialización de las escrituras*. Esto significa que si las operaciones de lectura del procesador P_1 a una localización ven el valor producido por la escritura w_1 (de, por ejemplo, P_2) antes que el valor producido por la escritura w_2 (de, por ejemplo, P_3), entonces las lecturas de otro procesador P_4 (o P_2 o P_3) no deben poder ver w_2 antes que w_1. No hay necesidad de un concepto análogo para la serialización de las lecturas, dado que los efectos de las lecturas no son visibles para cualquier procesador distinto de aquel que la realiza.

El resultado de un programa puede ser visto como los valores devueltos por las operaciones de lectura que realiza, quizás aumentado con un conjunto implícito de lecturas a todas las localizaciones al final del programa. De los resultados no podemos determinar el orden en el cual las operaciones se realizaron realmente por la máquina, sino el orden en el que parece que se ejecutaron. De hecho, no es importante en qué orden las cosas ocurren realmente en la máquina o cuándo cambió cada bit, dado que no es detectable; todo lo que importa es el orden en el cual las cosas parecen haber ocurrido, como detectable a partir de los resultados de una ejecución. Este concepto será más importante cuando discutamos los modelos de consistencia de la memoria. Para finalizar, una definición adicional que necesitamos en el caso de los multiprocesadores es la de completitud de una operación: Una operación de lectura o escritura se dice que *se ha completado* cuando se ha realizada con respecto a todos los procesadores.

5.1.4. Esquemas de coherencia de las cachés

Existen numerosos protocolos para mantener la coherencia de las cachés en un sistema multiprocesador con memoria compartida. En primer lugar hay dos formas de abordar el problema. Una forma es resolver el problema de la coherencia por software, lo que implica la realización de compiladores que eviten la incoherencia entre cachés de datos compartidos. La otra aproximación es proveer mecanismos hardware que mantengan de forma continua la coherencia en el sistema, siendo además transparente al programador. Como esta segunda forma es la más utilizada, nos centraremos en ella de manera que todos los protocolos que siguen se refieren a soluciones hardware al problema de la coherencia.

Podemos distinguir también dos tipos de sistemas multiprocesadores; por un lado están los sistemas basados en un único bus, con un número no demasiado grande de procesadores, cada uno con su caché, y por otro lado están los sistemas más complejos con varios buses o varias subredes con un gran número de nodos procesadores. En el primero parece más adecuado un tipo de protocolo que esté continuamente sondeando o *fisgando* el bus común para ver qué transacciones podrían introducir incoherencia y actuar en consecuencia. A estos protocolos se les llama de sondeo o *snoopy* puesto que fisgan el bus para detectar incoherencia. Básicamente cada nodo procesador tendrá los bits necesarios para indicar el estado de cada línea de su caché y así realizar las transacciones de coherencia necesarias según lo que ocurra en el bus en cada momento.

En el segundo tipo de sistemas, con varias subredes locales y un amplio número de procesadores, un protocolo de sondeo es complicado de realizar puesto que las actividades en los sub-buses son difíciles de fisgar, ya que las transacciones van a ser locales a estos. Para estos sistemas más complejos se utiliza un tipo de protocolo basado en directorio, que consiste en la existencia de un directorio común donde se guardan el estado de validez de las líneas de las cachés, de manera que cualquier nodo puede acceder a este directorio común.

Dado que los sistemas multiprocesadores basados en memoria compartida y caché suelen estar basados en bus, o una jerarquía de buses con no muchos niveles, los protocolos de coherencia suelen ser de sondeo, por esto se le dedica una atención especial a este tipo de protocolos.

Entre los protocolos de coherencia, tanto si son de sondeo como si no, existen en general dos políticas para mantener la coherencia: *invalidación en escritura* (*write invalidate*) y *actualización en escritura* (*write update*). En la política de invalidación en escritura (también llamada política de coherencia dinámica), siempre que un procesador modifica un dato de un bloque en la caché, invalida todas las demás copias de ese bloque que guardadas en las otras cachés. Por contra, la política de actualización en escritura actualiza las copias existentes en las otras cachés en vez de invalidarlas.

En estas políticas, los protocolos usados para la invalidación o actualización de las otras copias dependen de la red de interconexión empleada. Cuando la red de interconexión permite el *broadcast* (como en un bus), los comandos de invalidación y actualización pueden ser enviados a todas las cachés de forma simultánea. A los protocolos que utilizan esta técnica se les denominan, tal y como se ha comentado antes, *protocolos de de sondeo o* snoopy, dado que cada caché *monitoriza* las transacciones de las demás cachés. En otras redes de interconexión, donde el *broadcast* no es posible o causa degradación (como en las redes multietapa), el comando de invalidación/actualización se envía únicamente a aquellas cachés que tienen una copia del bloque. Para hacer esto, se usa frecuentemente un directorio centralizado o distribuido. Este directorio tiene una entrada para cada bloque. La entrada por cada bloque contiene un puntero a cada caché que tiene una copia del bloque. También contiene un bit que especifica si el permiso de actualizar el bloque sólo lo tiene una determinada caché. A los protocolos que usan este esquema se les denominan *protocolos de caché basados en directorio.*

5.2. Protocolos de sondeo o *snoopy* (medio compartido)

Tal y como se ha comentado antes, hay dos tipos de protocolos atendiendo al mecanismo que utilizan. Así se tienen el *write-update* o *write-broadcast* o *actualizar en escritura*, y el *write-invalidate* o *invalidar en escritura.*

El primer método, el de **invalidar en escritura**, se basa en asegurar que un procesador tiene acceso exclusivo a un dato antes de que acceda a él. Esto se consigue invalidando todas las líneas de todas las cachés que contengan un dato que está siendo escrito en ese momento. Este protocolo es con diferencia el más usado tanto en protocolos de sondeo como en los de directorio. El acceso exclusivo asegura que no hay otras copias leíbles o escribibles del dato cuando se realiza la escritura, ya que todas las copias del dato se invalidan en el momento de la escritura. Esto asegura que cuando otro procesador quiera leer el dato falle su caché y tenga que ir a memoria principal a buscarlo.

Si dos procesadores intentan escribir a la vez el mismo dato, hay uno que ganará necesariamente, mientras que el otro deberá esperar a que este escriba momento en el cual deberá volver a obtener la copia buena para poder volver a escribir. Por lo tanto, este protocolo asegura la escritura en serie.

El segundo método, alternativa al anterior, es el de actualizar todas las copias de

las cachés cuando se escribe un determinado dato. Este protocolo se llama *actualizar en escritura*. Para mantener los requisitos de ancho de banda de este protocolo bajo control, es interesante realizar un seguimiento de si una palabra en la caché está compartida o no, o sea, que se encuentra también en otras cachés. Si no está compartida entonces no es necesario actualizar el resto de cachés.

La belleza de la coherencia de caché basada en *snooping* reside en que toda la maquinaria para resolver un problema complicado se reduce a una pequeña cantidad de interpretación extra de eventos que ocurren de forma natural en el sistema. El procesador permanece inalterado. No existen operaciones explícitas de coherencia insertadas en el programa. Extendiendo los requisitos del controlador de la caché y explotando las propiedades del bus, las lecturas y escrituras que son inherentes al programa se usan de forma implícita para mantener la coherencia de las cachés mientras que la serialización del bus mantiene la consistencia. Cada controlador de caché observa e interpreta las transacciones de memoria de los otros controladores para mantener su propio estado interno. Para conseguir un uso eficiente de ancho de banda limitado que proporciona un bus compartido nos centraremos en protocolos que usan cachés del tipo post-escritura (*write-back*) lo que permite que varios procesadores escriban a diferentes bloques en sus cachés locales de forma simultánea sin ninguna transacción en el bus. En este caso es necesario tener especial cuidado para asegurarnos de que se transmite la suficiente información sobre el bus para poder mantener la coherencia. También comprobaremos que los protocolos proporcionan restricciones suficientes en la ordenación para mantener la serialización de las escrituras y un modelo de consistencia de memoria secuencial.

Recordemos que en un uniprocesador con una caché con post-escritura. Un fallo de escritura en la caché hace que ésta lea todo el bloque de la memoria, actualice una palabra, y retenga el bloque como *modificado* de tal forma que la actualización en memoria principal ocurre cuando el bloque es reemplazado. En un multiprocesador, este estado de modificado también se usa por parte del protocolo para indicar la pertenencia de forma exclusiva del bloque por parte de la caché. En general, se dice que una caché es *propietaria* de un bloque si debe suministrar los datos ante una petición de ese bloque. Se dice que una caché tiene una copia *exclusiva* de un bloque si es la única caché con una copia válida del bloque (la memoria principal puede o no tener una copia válida). La exclusividad implica que la caché puede modificar el bloque sin notificárselo a nadie. Si una caché no tiene la exclusividad, entonces no puede escribir un nuevo valor en el bloque antes de poner un transacción en el bus para comunicarse con las otras cachés. Si una caché tiene el bloque en estado modificado, entonces dicho nodo es el propietario y tiene la exclusividad. (La necesidad de distinguir entre propiedad y exclusividad se aclarará muy pronto.)

En un fallo de escritura, una forma especial de transacción llamada *lectura exclusiva* se usa para impedir a las otras cachés la escritura o adquirir una copia del bloque con propiedad exclusiva. Esto hace que el bloque de la caché pase a estado modificado permitiendo la escritura. Varios procesadores no pueden escribir el mismo bloque de forma concurrente, lo que llevaría a valores inconsistentes: Las transacciones de lectura exclusiva generadas por sus escrituras aparecerán de forma secuencial en el bus, así que sólo una de ellas puede obtener la propiedad exclusiva del bloque en un momento dado. Las acciones de coherencia de caché se consiguen mediante estos dos tipos de transacciones: lectura y lectura exclusiva. Eventualmente, cuando un bloque modificado se reemplaza en la caché, los datos se actualizan en la memoria, pero este evento no está producido por una operación de memoria sobre ese bloque y su importancia en el protocolo es incidental. Un bloque que no ha sido modificado no tiene que se escrito en

la memoria y puede ser descartado.

También debemos considerar los protocolos basados en la actualización. En este caso, cuando se escribe en una localización compartida, el valor actualizado se transfiere a todos los procesadores que tienen ese bloque en la caché. De esta forma, cuando los procesadores acceden a ese bloque posteriormente, pueden hacerlo desde sus cachés con baja latencia. Las cachés de todos los demás procesadores se actualizan con una única transacción del bus, conservando el ancho de banda. Por contra, con los protocolos basados en la invalidación, en una operación de escritura el estado de la caché de ese bloque de memoria en todos los demás procesadores son declarados inválidos. Las ventajas e inconvenientes de ambos protocolos son complejos y dependen de la carga de trabajo. En general, las estrategias basadas en la invalidación se han mostrado más robustas y se proporcionan como el protocolo básico por la mayoría de los vendedores. Algunos vendedores proporcionan un protocolo de actualización como una opción a ser utilizada para bloques correspondientes a estructuras de datos específicas o páginas.

La tabla 5.1 resume los principales protocolos e implementaciones para mantener la coherencia en sistemas basados en bus. El número de protocolos del tipo de invalidar en escritura es mayor puesto que son los protocolos que se utilizan más.

Nombre	Tipo de protocolo	Escritura a memoria	Características	Máquinas
Write once	Write invalidate	Write back después de la primera escritura	Primer protocolo de sondeo	Sequent Symmetry
Synapse N+1	Write invalidate	Write back	Estado explícito donde la memoria es propietaria	Máquinas Synapse; primeras disponibles
Berkeley	Write invalidate	Write back	Estado propietario compartido	Berkeley SPUR
Illinois MESI	Write invalidate	Write back	Estado privado limpio; puede entregar datos desde cualquier caché con copias limpias	SGI Power y series Challenge
Firefly	Write update	Write back si privado, write through si compartido	Memoria actualizada para todos	Actualmente en desuso; SPARCCenter 2000
Dragon	Write update	Write back si privado, write through si compartido		

Cuadro 5.1: Protocolos de sondeo para la coherencia de cachés.

A continuación veremos los protocolos MSI, MESI (conocido también como Illinois), Write once y Berkeley como ejemplos de protocolos basados en invalidación en escritura, y Dragon y Firefly que se incluyen como ejemplos de protocolos de actualización en escritura.

5.2.1. Protocolo de invalidación de 3 estados (MSI)

El primer protocolo que consideraremos es un protocolo de invalidación básico para cachés post-escritura. Es muy similar al protocolo que fue usado en la serie de multiprocesadores Silicon Graphics 4D. El protocolo usa los tres estados necesarios en cualquier caché post-escritura para distinguir bloques válidos que no han sido modificados (*clean*) de aquellos que han sido modificados (*dirty*). Específicamente, los estados son *inválido* (I), *compartido* (S) y *modificado* (M). El estado de inválido tiene un significado claro. Compartido significa que el bloque está presente en la caché y no ha sido modificado, la memoria principal está actualizada y cero o más cachés adicionales pueden tener también una copia actualizada (compartida). Modificado significa que únicamente este procesador tiene una copia válida del bloque en su caché, la copia en la memoria principal está anticuada y ninguna otra caché puede tener una copia válida del bloque (ni en estado modificado ni compartido). Antes de que un bloque compartido pueda ser escrito y pasar al estado modificado, todas las demás copias potenciales deben de ser invalidadas vía una transacción de bus de lectura exclusiva. Esta transacción sirve para ordenar la escritura al mismo tiempo que causa la invalidaciones y por tanto asegura que la escritura se hace visible a los demás.

El procesador emite dos tipos de peticiones: lecturas (PrRd) y escrituras (PrWr). Las lecturas o escrituras pueden ser a un bloque de memoria que existe en la caché o a uno que no existe. En el último caso, el bloque que esté en la caché en la actualidad será reemplazado por el nuevo bloque, y si el bloque actual está en el estado modificado su contenido será volcado a la memoria principal.

Supongamos que el bus permite las siguientes transacciones:

- Lectura del bus (BusRd): El controlador de la caché pone la dirección en el bus y pide una copia que no piensa modificar. El sistema de memoria (posiblemente otra caché) proporciona el dato. Esta transacción se genera por un PrRd que falla en la caché y el procesador espera como resultado de esta transacción el dato solicitado.
- Lectura exclusiva del bus (BusRdX): El controlador de la caché pone la dirección en el bus y pide una copia exclusiva que piensa modificar. El sistema de memoria (posiblemente otra caché) proporciona el dato. Todas las demás cachés necesitan ser invalidadas. Esta transacción se genera por una PrWr a un bloque que, o no está en la caché o está en la caché en un estado distinto al modificado. Una vez la caché obtiene la copia exclusiva, la escritura puede realizarse en la caché. El procesador puede solicitar una confirmación como resultado de esta transacción.
- Escritura (BusWB): El controlador de la caché pone la dirección y el contenido para el bloque de la memoria en el bus. La memoria principal se actualiza con el último contenido. Esta transacción la genera el controlador de la caché en una post-escritura; el procesador no conoce este hecho y no espera una respuesta.

La lectura exclusiva del bus es una nueva transacción que puede no existir excepto para la coherencia de la caché. Otro nuevo concepto necesario para soportar los protocolos de post-escritura es que junto con el cambio de estado de cada bloque en la caché, el controlador de la caché puede intervenir en las transacciones del bus y poner el contenido del bloque referenciado en el bus, en vez de permitir que la memoria proporcione el dato. Por supuesto, el controlador de la caché también puede iniciar nuevas transacciones, proporcionar datos para las post-escrituras, o coger los datos proporcionados por el sistema de memoria.

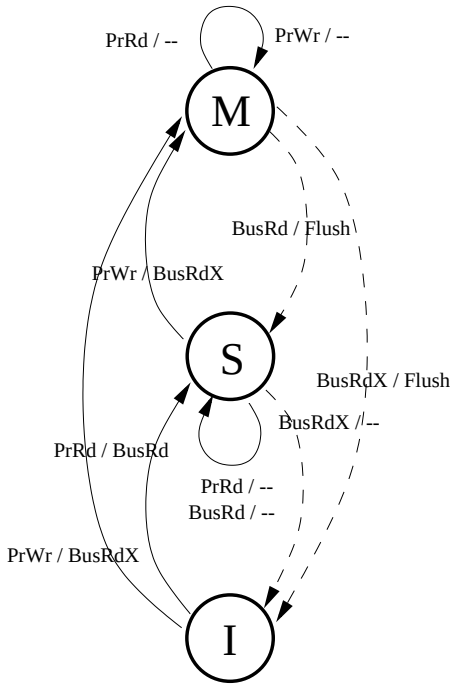


Figura 5.4: Protocolo básico de invalidación con tres estados.

El diagrama de estados que gobierna un bloque en cada caché en este protocolo se muestra en la figura 5.4. Los estados están organizados de tal manera que conforme un estado está más cerca de la parte superior más ligado está el bloque a ese procesador. La lectura de un bloque inválido por parte de un procesador (incluyendo los no presentes) produce una transacción BusRd. El bloque recientemente cargado se promociona desde el estado inválido al estado compartido dependiendo de si otra caché tiene una copia del bloque. Cualquier otra caché con el bloque en el estado compartido observa el BusRd, pero no toma ninguna acción especial, permitiendo que la memoria responda con el dato. Sin embargo, si una caché tiene el bloque en el estado modificado (y únicamente puede haber una) y observa una transacción BusRd en el bus, debe participar en la transacción ya que la copia existente en memoria no es válida. Esta caché envía los datos al bus en lugar de la memoria y degrada su copia del bloque al estado compartido. La memoria y la caché que pedía el dato son actualizadas. Esto puede conseguirse mediante una transferencia directa entre las cachés a través del bus durante esta transacción BusRd o indicando un error en la transacción BusRd y generando una transacción de escritura para actualizar la memoria. En el último caso, la caché original volvería a realizar su solicitud para obtener el bloque de la memoria. (También es posible hacer que el dato puesto en el bus sea tomado por la caché que realizó la petición pero no por la memoria, dejando la memoria sin actualizar, pero este caso requiere más estados).

Una escritura en un bloque inválido implica cargar el bloque entero y después modificar los bytes que se deseen. En este caso se genera una transacción de lectura exclusiva en el bus, que causa que todas las otras copias del bloque sean invalidadas, asegurando a la caché que realiza la petición la propiedad exclusiva del bloque. El bloque asciende al estado de modificado y después se escribe. Si otra caché realiza una petición posterior de acceso exclusivo, en respuesta a su transacción BusRdX el bloque se degradará al estado de invalido después de poner la copia exclusiva en el bus.

La transacción más interesante ocurre cuando se escribe en un bloque compartido. El tratamiento en este caso es igual al de una escritura en un bloque inválido, usando una transacción de lectura exclusiva en el bus para adquirir la propiedad exclusiva del bloque. Los datos que nos devuelve el bus pueden ser ignorados en este caso ya que ya existen en la caché. De hecho, una optimización común para reducir el tráfico en el bus es introducir una nueva transacción, llamada *actualización del bus* o BusUpgr, para esta situación. Un BusUpgr obtiene la propiedad exclusiva al igual que un BusRdX, invalidando las demás copias, pero no devuelve el bloque al que realizó la petición. Independientemente de la transacción utilizada, el bloque pasa al estado modificado. Posteriores escrituras en el bloque mientras permanece en ese estado no generan nuevas transacciones.

El reemplazamiento de un bloque de la caché lleva a dicho bloque al estado de inválido (no presente) eliminándolo de la caché. Por lo tanto, un reemplazamiento supone que dos bloques cambien de estado en la caché: el que es reemplazado pasa de su estado actual al estado de inválido, el que ocupa su lugar de inválido a su nuevo estado. El último cambio no puede tener lugar antes del primero. Si el bloque a ser reemplazado estaba en el estado de modificado, la transición de reemplazamiento de M a I genera una transacción de post-escritura. Si estaba en el estado inválido o compartido, no es necesario hacer nada.

El resultado de aplicar el protocolo MSI al ejemplo mostrado en la figura 5.2 se puede ver en la figura 5.5.

Acción Proces.	Est. P_1	Est. P_2	Est. P_3	Acc. bus	Datos sumin. por
1. P_1 lee u	S	–	–	BusRd	Memoria
2. P_3 lee u	S	–	S	BusRd	Memoria
3. P_3 escribe u	I	–	M	BusRdX	Memoria
4. P_1 lee u	S	–	S	BusRd	Caché de P_3
5. P_2 lee u	S	S	S	BusRd	Memoria

Figura 5.5: Protocolo de invalidación de 3 estados en acción para las transacciones mostradas en la figura 5.2.

Satisfacción de la coherencia

Dado que pueden existir lecturas y escrituras a cachés locales de forma simultánea, no es obvio que se satisfaga la condición de coherencia y mucho menos la de consistencia secuencial. Examinemos en primer lugar la coherencia. Las transacciones de lectura exclusiva aseguran que la caché en donde se realiza la escritura tiene la única copia válida cuando el bloque se modifica en la caché. El único problema que podría existir es que no todas las escrituras generan transacciones en el bus. Sin embargo, la clave aquí es que entre dos transacciones de bus para un bloque, únicamente un procesador puede realizar las escrituras; el que realizó la transacción de lectura exclusiva más reciente. En la serialización, esta secuencia de escrituras aparecerá en el orden indicado por el programa entre esa transacción y la siguiente transacción para ese bloque. Las lecturas que realice dicho procesador también aparecerán en orden con respecto a las otras escrituras. Para una lectura de otro procesador, existirá al menos una transacción

del bus que separará la finalización de la lectura de la finalización de esas escrituras. Esa transacción en el bus asegura que la lectura verá todas las escrituras en un orden secuencial consistente. Por lo tanto, las lecturas de todos los procesadores ven todas las escrituras en el mismo orden.

Satisfacción de la consistencia secuencial

Para comprobar que se cumple la consistencia secuencial, veamos en primer lugar cuál es la propia definición y cómo puede construirse una consistencia global entrelazando todas las operaciones de memoria. La serialización del bus, de hecho, define un orden total sobre las transacciones de todos los bloques, no sólo aquellas que afectan a un único bloque. Todos los controladores de cachés observan las transacciones de lectura y lectura exclusiva en el mismo orden y realizan las invalidaciones en ese orden. Entre transacciones consecutivas del bus, cada procesador realiza una secuencia de operaciones a la memoria (lecturas y escrituras) en el orden del programa. Por lo tanto, cualquier ejecución de un programa define un orden parcial natural:

> Una operación de memoria M_j es posterior a la operación M_i si (i) las operaciones son emitidas por el mismo procesador y M_j aparece después que M_i en el orden del programa, o (ii) M_j genera una transacción del bus posterior a la operación de memoria M_i.

Este orden parcial puede expresarse gráficamente como se muestra en la figura . Entre transacciones del bus, cualquiera de las secuencias que se pueden formar entrelazando las distintas operaciones da lugar a un orden total consistente. En un segmento entre transacciones del bus, un procesador puede observar las escrituras de otros procesadores, ordenadas por transacciones de bus previas que genera, al mismo tiempo que sus propias escrituras están ordenadas por el propio programa.

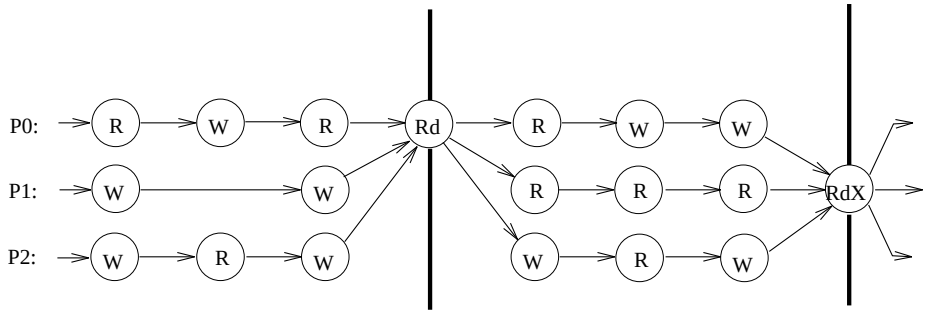


Figura 5.6: Orden parcial en las operaciones de memoria para una ejecución con el protocolo MSI.

También podemos ver cómo se cumple la CS en términos de las condiciones suficientes. La detección de la finalización de la escritura se realiza a través la aparición de una transacción de lectura exclusiva en el bus y la realización de la escritura en la caché. La tercera condición, que proporciona la escritura atómica, se consigue bien porque una lectura (i) causa una transacción en el bus que es posterior a aquella transacción de escritura cuyo valor desea obtener, en cuyo caso la escritura debe haberse completado de forma global antes de la lectura, o (ii) porque es posterior en el orden del programa

como es el caso de una lectura dentro del mismo procesador, o (iii) porque es posterior en el orden del programa del mismo procesador que realizó la escritura, en cuyo caso el procesador ya ha esperado el resultado de una transacción de lectura exclusiva y la escritura se ha completado de forma global. Así, se cumplen todas las condiciones suficientes.

5.2.2. Protocolo de invalidación de 4 estados (MESI)

El protocolo MSI presenta un problema si consideramos una aplicación secuencial en un multiprocesador; este uso de la multiprogramación es de hecho la carga más usual de multiprocesadores de pequeña escala. Cuando el programa lee y modifica un dato, el protocolo MSI debe generar dos transacciones incluso en el caso de que no exista compartición del dato. La primera es una transacción BusRd que obtiene el bloque de la memoria en estado S, y la segunda en una transacción BusRdX (o BusUpgr) que convierte el bloque del estado S al M. Añadiendo un estado que indique que el bloque es la única copia (exclusiva) pero que no está modificado y cargado el bloque en ese estado, podríamos evitarnos la última transacción, ya que el estado indica que ningún otro procesador tiene el bloque en caché. Este nuevo estado indica un nivel intermedio entre compartido y modificado. Al ser exclusivo es posible realizar una escritura o pasar al estado modificado sin ninguna transacción del bus, al contrario que en el caso de estar en el estado compartido; pero no implica pertenencia, así que al contrario que en el estado modificado la caché no necesita responder al observar una petición de dicho bloque (la memoria tiene una copia válida). Variantes de este protocolo MESI se usan en muchos de los microprocesadores modernos, incluyendo el Pentium, PowerPC 601 y el MIPS R4400 usado en los multiprocesadores Silicon Graphics Challenge. A este protocolo también se le conoce como protocolo Illinois por haber sido publicado por investigadores de la Universidad de Illinois en 1984.

El protocolo MESI consiste en cuatro estados: *modificado* (M), *exclusivo* (E), *compartido* (S) e *inválido* (I). I y M tienen la misma semántica que antes. El estado exclusivo, E, significa que únicamente una caché (esta caché) tiene una copia del bloque, y no ha sido modificado (es decir, la memoria principal está actualizada). El estado S significa que potencialmente dos o más procesadores tienen este bloque en su caché en un estado no modificado.

Cuando un procesador lee por primera vez un bloque, si existe una copia válida en otra caché entra en la caché del procesador con el estado S. Sin embargo, si no existe otra copia en ese instante (por ejemplo, en una aplicación secuencial), entra con el estado E. Cuando ese bloque se actualiza por medio de una escritura puede pasar directamente del estado E a M sin generar otra transacción en el bus, dado que no existe otra copia. Si mientras tanto otra caché ha obtenido una copia del bloque, el estado del bloque deberá pasar de E a S. Este protocolo necesita que el bus proporcione una señal adicional (S) que esté disponible para que los controladores puedan determinar en una transacción BusRd si existe otra caché que tenga el mismo bloque. Durante la fase en la que se indica en el bus la dirección del bloque al que se quiere acceder, todas las cachés determinan si tienen el bloque que se solicita y, si eso ocurre, activan la señal compartida. Esta es una línea cableada como un OR, con lo que el controlador que realiza la petición puede observar si existe otro procesador que tenga en su caché el bloque de memoria referenciado y decidir si la carga del bloque se debe realizar en el estado S o E.

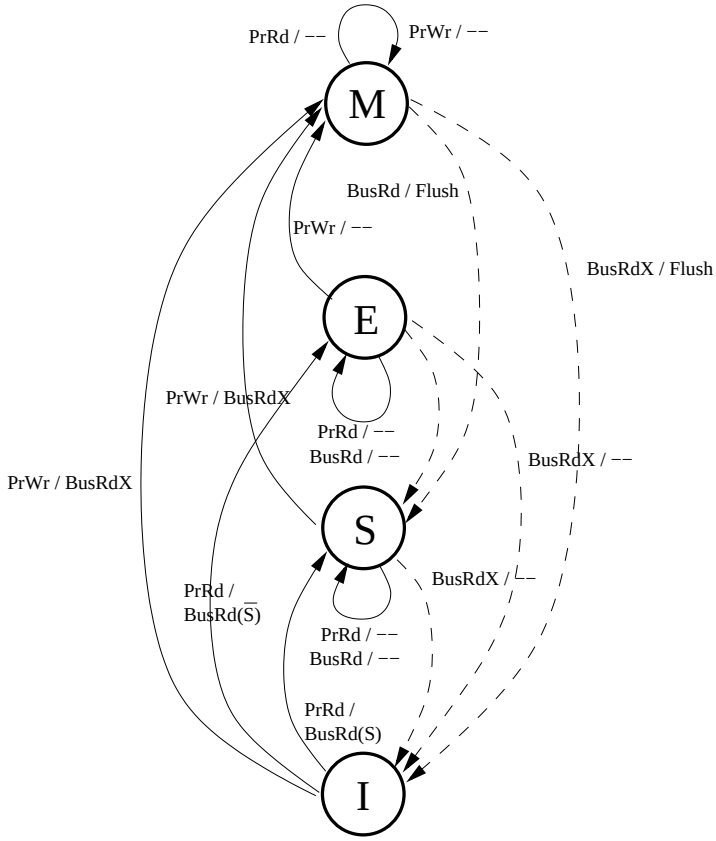


Figura 5.7: Diagrama de transición de estados para el protocolo Illinois MESI.

El diagrama de transición de estados para este protocolo se muestra en la figura 5.7. La notación BusRd(S) significa que cuando la transacción de lectura en el bus ocurre, se activa la señal S, y BusRd($\overline{S}$) significa que S no está activada. Una transacción BusRd significa que no nos importa el valor de S en esa transacción. Una escritura a un bloque en cualquier estado hace que este pase al estado M, pero si estaba en el estado E no se necesita una transacción del bus. La aparición de un BusRd hace que el bloque pase del estado E al S, dado que existe otra copia del bloque. Como ocurría anteriormente, la aparición de una transacción BusRd hace que un bloque que estuviese en el estado M pase al estado S y dicho bloque sea puesto en el bus. Tal como ocurría en el protocolo MSI es posible hacer que dicho bloque no sea actualizado en la memoria principal, pero serían necesarios nuevos estados. Obsérvese que es posible que un bloque esté en el estado S incluso sin existir otras copias, dado que estas copias pueden ser reemplazadas sin que las demás cachés sean notificadas. Los argumentos para comprobar que el protocolo mantiene la consistencia secuencial y la coherencia de las cachés es el mismo que en el caso del protocolo MSI.

El nombre del protocolo (MESI) viene de los nombres de los estados que puede tomar una línea: *modificada*, *exclusiva*, *compartida* e *inválida* que en inglés significan *Modified*, *Exclusive*, *Shared* y *Invalid*, y lo cierto es que este nombre se usa más que el de Illinois.

Una línea que es propiedad de cualquier caché puede estar como *exclusiva* o *modificada*. Las líneas que están en el estado *compartida* no son propiedad de nadie en concreto. La aproximación MESI permite la determinación de si una línea es compartida o no

cuando se carga. Una línea modificada es enviada directamente al peticionario desde la caché que la contiene. Esto evita la necesidad de enviar comandos de invalidación para todas aquellas líneas que no estaban compartidas.

Veamos las transiciones según las operaciones realizadas sobre las cachés:

Fallo de lectura: La línea se coge de la caché que la tenga. Si ninguna la tiene, se coge de la memoria principal. Si la línea que se coge está *modificada* entonces se escribe esta línea también a la memoria al mismo tiempo. Si es compartida, la caché con mayor prioridad es la que entrega la línea. La caché que entrega la línea, y aquellas que la contienen, pasan al estado de *compartida*. Sin embargo, si quien da la línea es la memoria, entonces la caché que lee pone su línea en *exclusiva* y se hace propietaria de ella.

Fallo de escritura: En este caso la línea se coge de la caché que la tenga. Si ninguna la tiene, se coge de la memoria principal. Si la línea que se coge está *modificada* entonces se escribe esta línea también a la memoria al mismo tiempo. Si es compartida, la caché con mayor prioridad es la que entrega la línea. La línea leída se actualiza y se pone en estado *modificada*. El resto de líneas pasan al estado *invalidas*. Hay que hacer notar que este protocolo sólo permite un único escribiente, es decir, una línea modificada sólo está presente en una única caché.

Acierto de escritura: Si la caché peticionaria es la propietaria (línea *modificada* o *exclusiva*), se actualiza la línea en la caché sin más. Si la línea está compartida por otras cachés (*compartida*), se actualiza la línea después de haber invalidado el resto de líneas. Una vez actualizada la línea pasa a ser *modificada* en cualquiera de los casos anteriores.

5.2.3. Write once

A veces se le conoce también como *write invalidate* directamente, ya que es el primero que se pensó de este tipo y también es el primero de los protocolos de sondeo. En ocasiones se le conoce como el protocolo de Goodman puesto que fue propuesto por James Goodman en 1983.

Este protocolo de coherencia se puede describir mediante un grafo de transiciones de 4 estados tal y como se muestra en la figura 5.8. Los 4 estados son los siguientes:

Válida: La línea de caché, que es consistente con la copia en memoria, ha sido leída de la memoria principal y no ha sido modificada.

Inválida: La línea no se encuentra en la caché o no es consistente con la copia en memoria.

Reservada: Los datos han sido escritos una única vez desde que se leyó de la memoria compartida. La línea de caché es consistente con la copia en memoria que es la única otra copia.

Sucia: La línea de caché ha sido escrita más de una vez, y la copia de la caché es la única en el sistema (por lo tanto inconsistente con el resto de copias).

Cada línea de caché tiene dos bits extra donde se guarda el estado de esa línea. Dependiendo de la bibliografía cambia el nombre de los estados. Así a la línea *válida* se la conoce también como *consistente múltiple*, a la *reservada* como *consistente única*, y la *sucia* como *inconsistente única*, la *válida* se queda con el mismo nombre.

En realidad este protocolo de *escribir una vez* está basado en un protocolo más

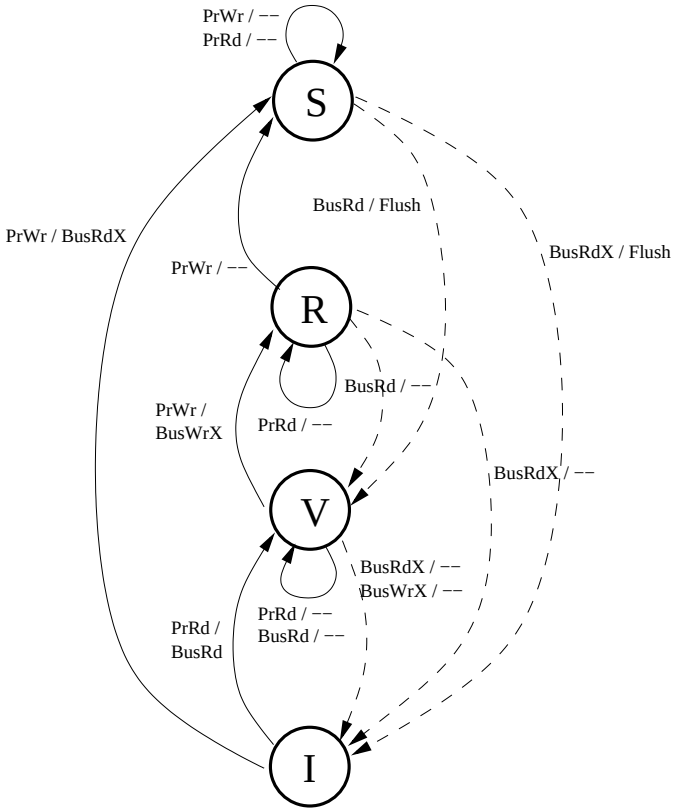


Figura 5.8: Diagrama de transición del protocolo Write once.

sencillo con sólo tres estados para las copias que son *válida*, *inválida* y *sucia*. El estado *reservada* se ha añadido para el caso en que se modifique una línea una sola vez, de manera que se aprovecha mejor el ancho de banda del bus.

Para mantener la consistencia el protocolo requiere dos conjuntos de comandos. Las líneas sólidas de la figura 5.8 se corresponden con los comandos emitidos por el procesador local y son *fallo de lectura*, *acierto de escritura* y *fallo de escritura*.

Siempre que se produce un fallo de lectura la línea entra en el estado de *válida*. El primer acierto de escritura lleva a la línea a su estado de *reservada*. Los aciertos de escritura posteriores cambian el estado a *sucia*.

Las líneas discontinuas se corresponden con los comandos de invalidación emitidos por el resto de procesadores a través del bus de sondeo compartido. El comando *invalidar en lectura* lee una línea o bloque e invalida el resto de copias. El comando *invalidar en escritura* invalida todas las copias de una línea. El comando *lectura de bus* se corresponde con una lectura normal a memoria por otro procesador.

Con estas consideraciones, los eventos y acciones que tienen lugar en el sistema de memoria son:

Fallo de lectura: Cuando un procesador quiere leer un bloque que no está en la caché se produce un *fallo de lectura*. Entonces se inicia una operación de *lectura en el bus*. Si no hay copias de esta línea en otras cachés entonces se trae la línea de la memoria principal, pasando el estado a *exclusiva* puesto que es la única caché con esa línea. Si hay alguna copia *exclusiva* en alguna otra caché, entonces se coge la copia de la memoria o de esa caché, pasando ambas copias al

estado de *válidas*. Si existe alguna copia *sucia* en otra caché, esa caché inhibirá la memoria principal y mandará esa copia a la caché que hizo la petición así como a la memoria principal, ambas copias resultantes pasarán al estado de *válidas*.

Acierto de escritura: Si la copia está *sucia* o *reservada*, la escritura se puede llevar a cabo localmente y el nuevo estado es *sucia*. Si el estado es *válida* entonces se manda un comando de *invalidar en escritura* a todas las cachés invalidando sus copias. A la memoria compartida se escribe al tiempo que a la propia caché, pasando el estado a *reservada*. De esta manera nos aseguramos de que sólo hay una copia que es *reservada* o *sucia* de una determinada línea en todas las cachés.

Fallo en escritura: Cuando un procesador falla al escribir en la caché local, la copia tiene que venir de la memoria principal o de la caché que esté sucia. Esto se realiza enviando un *invalidar en lectura* a todas las cachés lo que invalidará todas las cachés. La copia local es actualizada terminando en un estado de *sucia* puesto que no se ha actualizado la memoria después de la escritura.

Acierto de lectura: Nunca hay una transición en este caso, se lee de la caché local y ya está.

Cambio de línea: Si una copia está *sucia* debe escribirse en algún momento a la memoria principal mediante un cambio de línea (lo cual ocurrirá cuando alguna otra caché lea de esa línea). Si la copia esta *limpia*, es decir, es *válida*, *exclusiva* o *inválida*, no se realiza ningún cambio.

Para poder implementar este protocolo en un bus, son necesarias unas líneas adicionales para inhibir la memoria principal cuando se produce un fallo de lectura y hay copias sucias. Muchos de los buses estándar no incluyen estas líneas por lo que resulta complicado implantar protocolos de coherencia de caché, a pesar de que estos buses incluyen multiprocesadores.

5.2.4. Berkeley

El protocolo Berkeley, también llamado Berkeley-SPUR, usa la idea de propietario de la línea de caché. En cualquier instante una línea de caché sólo puede ser propiedad de una sola de las cachés, y si ninguna tiene esa línea entonces se encuentra en memoria principal. Hay cuatro estados para realizar esto: *invalida*, *sólo lectura*, *sucia compartida* y *sucia privada*. Cuando una línea está compartida, sólo el propietario tiene esa línea en el estado *sucia compartida*; todos los demás deberán tener esa línea como *sólo lectura*. Por lo tanto, una línea de caché sólo puede estar en el estado *sucia compartida* o *sucia privada* en una única caché, que será la propietaria de esa línea.

La figura 5.9 muestra el diagrama de estados correspondiente a este protocolo; a la izquierda se muestra el diagrama para las operaciones que realiza el procesador y a la derecha el diagrama para las órdenes que vienen a través del bus realizadas por el resto de procesadores.

Veamos a continuación las acciones a realizar dependiendo de las operaciones de la CPU:

Fallo de lectura: Cuando ocurre un *fallo de lectura*, la caché que falló pide los datos al propietario de la línea, que puede ser la memoria (si la línea es *inválida* o de *sólo lectura* en el resto de cachés) o la caché que sea propietaria de la línea (que tenga cualquiera de los estados sucios *compartida* o *privada*), pasando el estado de la línea a *sólo lectura*. Si la línea que se pide se encuentra en el estado de *sólo*

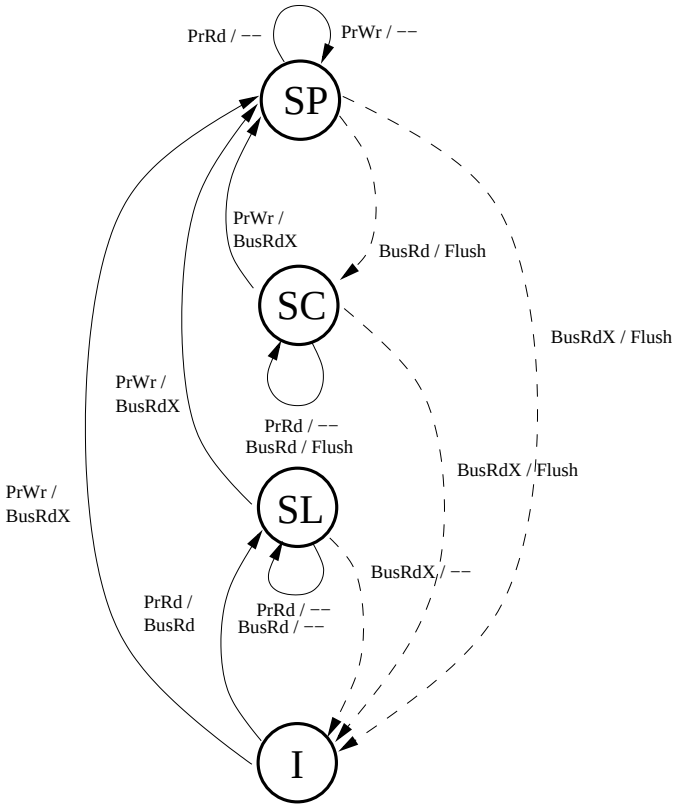


Figura 5.9: Diagramas de transición del protocolo Berkeley.

lectura en el resto de las cachés, entonces el dato lo coge de la memoria y el estado que se le asigna es el de *sólo lectura*. Si el propietario de la línea tenía el estado de *sucia privada* tendrá que cambiarlo al de *sucia compartida* puesto que a partir de ese momento existen más copias de esa línea en otras cachés.

Fallo de escritura: En un *fallo de escritura*, la línea viene directamente del propietario (memoria u otra caché). El resto de cachés con copias de la línea, incluida la inicialmente propietaria, deben invalidar sus copias. La caché que quería escribir pone el estado de esta línea a *sucia privada* y pasa a ser el propietario.

Acierto de escritura: La caché que escribe invalida cualquier otra copia de esa línea y actualiza la memoria. El nuevo estado de la línea se pone a *sucia privada* da igual el estado que tuviera esa línea. Sólo en el caso en que la línea fuera *sucia privada* no es necesaria ni la invalidación ni la actualización a memoria principal, con lo que la escritura se realiza local a la caché.

5.2.5. Protocolo de actualización de 4 estados (Dragon)

A continuación estudiaremos un protocolo de actualización básico para cachés de post-escritura. Esta es una versión mejorada del protocolo usado por los multiprocesadores SUN SparcServer. Este protocolo fue inicialmente propuesto por los investigadores de Xerox PARC para su sistema multiprocesador Dragon.

El protocolo Dragon consta de 4 estados: *exclusivo* (E), *compartido* (SC), *compartido modificado* (SM) y *modificado* (M). El estado exclusivo significa que sólo existe

una caché (esta caché) con una copia del bloque y que dicho bloque no ha sido modificado. La causa de añadir el estado E en Dragon es la misma que en el protocolo MESI. SC significa que potencialmente dos o más procesadores (incluyendo esta caché) tienen este bloque en su caché y que la memoria principal puede estar o no actualizada. SM significa que potencialmente dos o más procesadores tienen este bloque en su caché, la memoria principal no está actualizada y este procesador es el responsable de actualizar la memoria principal cuando este bloque sea reemplazado en la caché. Un bloque sólo puede estar en el estado SM en una única caché en un momento dado. Sin embargo, es posible que una caché tenga el bloque en estado SM mientras que en las otras aparezca en el estado SC. M significa, como anteriormente, que el bloque ha sido modificado únicamente en esta memoria, la memoria principal tiene un valor anticuado, y este procesador es el responsable de actualizar la memoria principal cuando el bloque sea reemplazado. Obsérvese que no existe un estado explícito de bloque inválido (I) como en los protocolos anteriores. Esto es debido a que el protocolo Dragon es un protocolo basado en la actualización; el protocolo siempre mantiene los bloque de la caché actualizados, así que siempre es posible utilizar los datos existentes en la caché.

Las peticiones del procesador, las transacciones del bus y las acciones a realizar por el protocolo son similares a las vistas en el protocolo Illinois MESI. El procesador todavía envía peticiones de lectura (PrRd) y escritura (PrWr). Sin embargo, dado que no tenemos un estado de invalidación en el protocolo, para especificar las acciones a realizar cuando un bloque de memoria se pide por primera vez, se han añadido dos tipos más de peticiones: *lectura con fallo de caché* (PrRdMiss) y *escritura con fallo de caché* (PrWrMiss). Para las transacciones de bus, tenemos *lectura en bus* (BusRd), *actualización en bus* (BusUpd) y *escritura en bus* (BusWB). Las transacciones BusRd y BusWB tienen la semántica usual vista en los protocolos anteriores. BusUpd es una nueva transacción que toma como parámetro la palabra modificada por el procesador y la envía a todos los procesadores a través del bus de tal manera que se actualicen las cachés de los demás procesadores. Enviando únicamente la palabra modificada en lugar de todo el bloque, se espera una utilización más eficiente del ancho del banda del bus. Al igual que en el caso del protocolo MESI aparece la señal S para soportar el estado E.

La figura 5.10 muestra el diagrama de transición de estados para este protocolo. A continuación se exponen las acciones a realizar cuando un procesador incurre en un fallo de caché en una lectura, un acierto de caché en una escritura o un fallo de caché en una escritura (en el caso de un acierto de caché en una lectura no hay que realizar ninguna acción).

Fallo en una lectura: Se genera una transacción BusRd. Dependiendo del estado de la señal compartida (S), el bloque se carga en el estado E o SC en la caché local. Más concretamente, si el bloque está en el estado M o SM en una de las otras cachés, esa caché activará la línea S y proporcionará el último valor para ese bloque en el bus, y el bloque se cargará en la caché local en el estado SC (también se actualiza la memoria principal; si no quisiéramos hacerlo necesitaríamos estados adicionales). Si la otra caché lo tiene en estado M, cambiará al estado SM. Si ninguna otra caché tiene una copia, la línea S no se activará y el dato será proporcionado por la memoria principal y cargado en la caché local en el estado E.

Escritura: Si el bloque está en el estado SM o M en la caché local, no es necesario realizar ninguna acción. Si el bloque está en el estado E en la

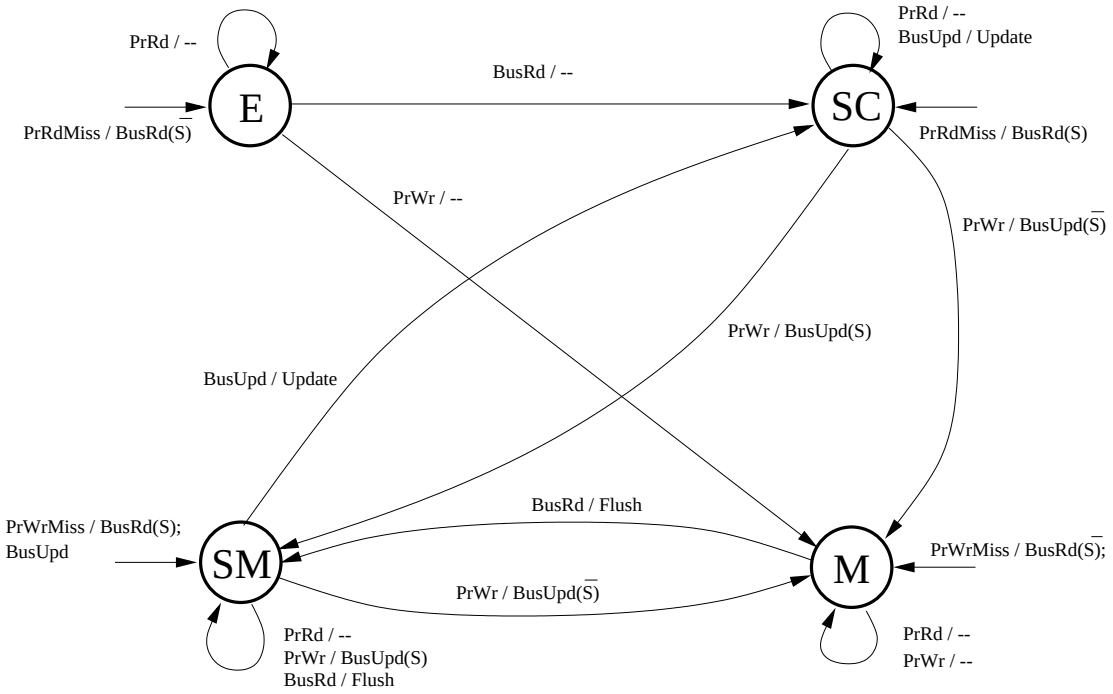


Figura 5.10: Diagrama de transición de estados del protocolo de actualización Dragon.

caché local, el bloque cambia internamente al estado M, no siendo necesaria ninguna acción adicional. Sin embargo, si el bloque está en el estado SC, se genera una transacción BusUpd. Si alguna otra caché tiene una copia del dato, actualizan sus copias y cambian su estado a SC. La caché local también actualiza su copia del bloque y cambia su estado a SM. Si no existe otra copia del dato, la señal S permanecerá inactiva y la copia local se actualizará cambiando al estado M. Finalmente, si en una escritura el bloque no está presente en la caché, la escritura se trata como una transacción del tipo fallo en la lectura seguida de una transacción de escritura. Por lo tanto, primero se genera un BusRd y si el bloque se también se encuentra en otras cachés un BusUpd.

Reemplazo: En el caso de reemplazo de un bloque (los arcos no se muestran en la figura), el bloque se escribe en la memoria principal utilizando una transacción del bus únicamente si se encuentra en el estado M o SM. Si está en el estado SC, existirá otra caché con ese bloque en estado SM, o nadie lo tiene en cuyo caso el bloque es válido en memoria principal.

5.2.6. Firefly

El Firefly tiene 3 estados que son: *lectura privada* os Exclusivo (E), *lectura compartida* o Compartida Consistente (SC) y *sucia privada* o Modificada (M). El estado de línea *inválida* se utiliza sólo para definir la condición inicial de una línea de caché. El Firefly utiliza el esquema de actualización en vez del de invalidación, es decir, las escrituras a la caché son vistas por todos y se escribe a la memoria principal. El resto de cachés que comparten la línea, fisgan el bus actualizando sus copias. Por lo tanto, ninguna línea de caché será inválida después de ser cargada. Hay una línea especial del

bus, la *LíneaCompartida*, que se activa para indicar que al menos otra caché comparte la línea.

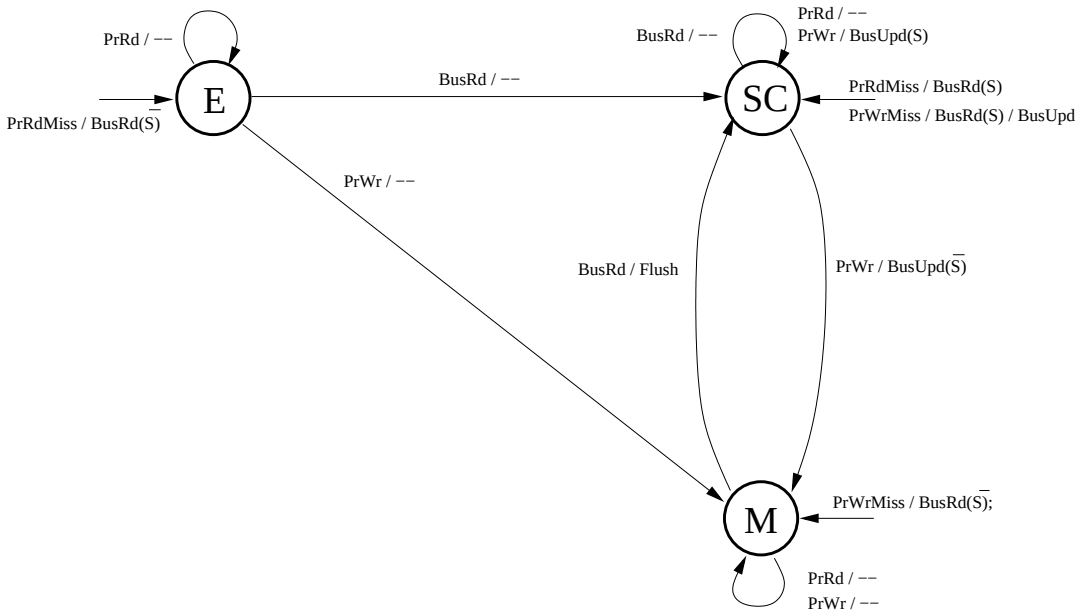


Figura 5.11: Diagrama de transición de estados del protocolo de actualización Firefly.

En la figura 5.11 se muestra el diagrama de transición de estados del protocolo Firefly. Veamos las transiciones a realizar según los casos:

Fallo de lectura: Si otra caché tiene una copia de la línea, ésta la entrega a la caché peticionaria, activando la línea *LíneaCompartida*. Si el estado de la línea era *sucia privada* previamente, entonces debe actualizarse a la memoria principal. Todas las cachés que comparten dicha línea ponen su estado a *lectura compartida*. Si no hay cachés con esta línea, entonces se coge de la memoria y se pone el estado de *lectura privada*.

Fallo de escritura: La línea se coge de otra caché o de la memoria. Cuando la línea viene de otra caché, ésta activa la línea *LíneaCompartida* que es reconocida por la caché peticionaria. Por lo tanto, la línea se pone en estado de *lectura compartida* y la escritura se propaga a todas las cachés y a memoria, que actualizan sus copias. Si la línea viene de la memoria, se carga la línea como *sucia privada* y se escribe sin propagar la línea con los nuevos datos.

Acierto en escritura: Si la línea está *privada sucia* o *lectura privada*, se escribe a la caché sin más. En el último caso se cambia el estado a *privada sucia*. Si el estado es el de *lectura compartida* entonces además de a la caché debe escribir a la memoria. El resto de cachés que comparten la línea capturan los datos del bus actualizando así sus líneas. Además, estas cachés activan la línea *LíneaCompartida*. Esto es necesario para que así la caché que está escribiendo pueda saber si la línea está compartida o no. Si la línea no está compartida, se puede evitar la escritura "a todos" (*broadcast*) aunque la transacción se debe realizar en cualquier caso. Si no está compartida el estado pasa a *modificada*, de otra manera se carga como *lectura compartida*.

5.2.7. Rendimiento de los protocolos de sondeo

A la hora de determinar el rendimiento de un multiprocesador con arquitecturas como las vistas hasta ahora, existen varios fenómenos que se combinan. En particular, el rendimiento total de la caché es una combinación del comportamiento de tráfico causado por fallos de caché en un uniprocesador y el trafico causado por la comunicación entre cachés, lo que puede dar lugar a invalidaciones y posteriores fallos de caché. El cambio del número de procesadores, tamaño de la caché y tamaño del bloque pueden afectar a estos dos componentes de diferente manera, llevando al sistema a un comportamiento que es una combinación de los dos efectos.

Las diferencias de rendimiento entre los protocolos de invalidación y actualización se basan en las tres características siguientes:

1. Varias escrituras a la misma palabra sin lecturas intermedias precisan varias escrituras a todos en el protocolo de actualización, pero sólo una escritura de invalidación en el de invalidación.
2. Con líneas de caché de varias palabras, cada palabra escrita en la línea precisa de una escritura a todos en el protocolo de actualización, mientras que sólo la primera escritura de cualquier palabra en la línea es necesaria en el de invalidación. Los protocolos de invalidación trabajan sobre bloques de caché, mientras que en el de actualización deben trabajar sobre palabras para aumentar la eficiencia.
3. El retraso entre la escritura de una palabra en un procesador y la lectura de ese valor escrito por otro procesador es habitualmente menor en un multiprocesador de actualización, debido a que los datos escritos son actualizados de forma inmediata en la caché del que lee (si es que el procesador que lee ya tenía copia de esa línea). Por comparación, en un protocolo de invalidación, el que va a leer se invalida primero, luego cuando llega la operación de lectura debe esperar a que esté disponible.

Como el ancho de banda del bus y la memoria es lo que más se demanda en un multiprocesador basado en bus, los protocolos de invalidación son los que más se han utilizado en estos casos. Los protocolos de actualización también causan problemas según los modelos de consistencia, reduciendo las ganancias de rendimiento potenciales mostradas en el punto 3 anterior. En diseños con pocos procesadores (2–4) donde los procesadores están fuertemente acoplados, la mayor demanda de ancho de banda de los protocolos de actualización puede ser aceptable. Sin embargo, dadas las tendencias de mayor rendimiento de los procesadores y la demanda consecuente en ancho de banda, hacen que los protocolos de actualización no se usen.

En un sistema que requiera mucha migración de procesos o sincronización, cualquier protocolo de invalidación va a funcionar mejor que los de actualización. Sin embargo, un fallo en la caché puede producirse por la invalidación iniciada por otro procesador previamente al acceso a la caché. Estos fallos por invalidación pueden incrementar el tráfico en el bus y deben ser reducidas.

Varias simulaciones han mostrado que el tráfico en el bus en un multiprocesador puede incrementarse cuando el tamaño de la línea se hace grande. El protocolo de invalidación facilita la implementación de primitivas de sincronización. Típicamente, la media de copias inválidas es bastante pequeña, como 2 o 3 en pequeños multiprocesadores.

El protocolo de actualización requiere la capacidad de realizar transacciones de

escritura "a todos" (*broadcast*). Este protocolo puede evitar el efecto "ping-pong" de los datos compartidos entre varias cachés. Reduciendo la cantidad de líneas compartidas se consigue una reducción del tráfico en el bus en un multiprocesador de este tipo. Sin embargo, este protocolo no puede ser usado con varias escrituras seguidas largas.

5.2.8. El problema de la falsa compartición

5.3. Esquemas de coherencia basados en directorio

Los protocolos de sondeo vistos con anterioridad precisan de arquitecturas que tengan la facilidad de acceso "a todos" a un tiempo; una arquitectura basada en bus es un buen ejemplo. El problema del bus es que no es una arquitectura que pueda acomodar un número elevado de procesadores debido a la limitación del ancho de banda. Para multiprocesadores con un número elevado de procesadores se utilizan otro tipo de interconexiones como las vistas en en el capítulo 5.6 como las mallas, hipercubos, etc. Sin embargo, estas redes más complejas no poseen la capacidad de que cualquier nodo pueda espiar lo que hace el resto. Para solucionar esto lo que se hace es introducir protocolos de coherencia basados en un directorio al cual todos los procesadores y cachés tienen acceso.

5.3.1. Protocolos basados en directorio

Veremos a continuación que los protocolos basados en directorio se pueden dividir en los que tienen el directorio centralizado y los que lo tienen distribuido. En ambos grupos se permite que existan varias copias compartidas de la misma línea de caché para mejorar el rendimiento del multiprocesador sin incrementar demasiado el tráfico en la red.

Directorio centralizado: Fue el primer esquema propuesto (1976). Consiste en un único directorio o tabla centralizada donde se guarda información sobre el lugar donde se encuentra cada copia de la caché. Este directorio centralizado es normalmente bastante grande por lo que la búsqueda se realiza de forma asociativa. La competencia por el acceso al directorio (contención), así como los largos tiempos de búsqueda, son alguno de los inconvenientes de este esquema. Los protocolos de directorio centralizado suelen ser o de *mapeado completo* o con *mapeado limitado* que son dos protocolos que se explican más adelante.

Directorio distribuido: Dos años después se propuso otro protocolo basado en la distribución del directorio entre las diferentes cachés. En el directorio se guarda el estado de la caché así como su presencia. El estado es local, pero la presencia indica qué cachés tienen una copia del bloque. También hay dos protocolos típicos de mapeado distribuido que son los protocolos con *directorios jerárquicos* y los protocolos con *directorios encadenados*.

En ocasiones la noción de directorio centralizado o distribuido crea confusión; si nuestro modelo de sistema tiene la memoria dividida en trozos, cada uno asociado a un nodo, entonces el directorio centralizado está en realidad distribuido entre las memorias. En este caso el directorio se encuentra tan distribuido como en un protocolo de directorio distribuido. La diferencia es que en el directorio centralizado, se accede a una porción

del directorio basándonos en su localización por su dirección física.

En el caso del directorio distribuido, se accede a un directorio particular porque ese nodo es propietario de la línea de caché.

Veamos a continuación los diferentes protocolos, de uno y otro tipo, más frecuentemente utilizados.

5.3.2. Protocolo de mapeado completo

El protocolo de mapeado completo mantiene un directorio en el cual cada entrada tiene un bit, llamado *bit de presencia*, por cada una de las cachés del sistema. El bit de presencia se utiliza para especificar la presencia en las cachés de copias del bloque de memoria. Cada bit determina si la copia del bloque está presente en la correspondiente caché. Por ejemplo, en la figura 5.12 las cachés de los procesadores P_a y P_c contienen una copia del bloque de datos x, pero la caché del procesador P_b no. Aparte, cada entrada del directorio tiene un bit llamado *bit de inconsistencia única*. Cuando este bit está activado, sólo uno de los bits de presencia está a uno, es decir, sólo existe una caché con la copia de ese bloque o línea, con lo que sólo esa caché tiene permiso para actualizar la línea.

Cada caché por su parte tiene dos bits en cada entrada de la caché. El primero es el *bit de validación* (v en la figura 5.12), e indica si la copia es válida o no. Si el bit es cero, indica que la copia no es válida, es decir, la copia se puede quitar de la caché. El otro bit, llamado *bit de privacidad* (p en la figura 5.12), sirve para indicar si la copia tiene permiso de escritura, es decir, cuando este bit es uno entonces es la única copia que existe de esta línea en las cachés y por tanto tiene permiso para escribir.

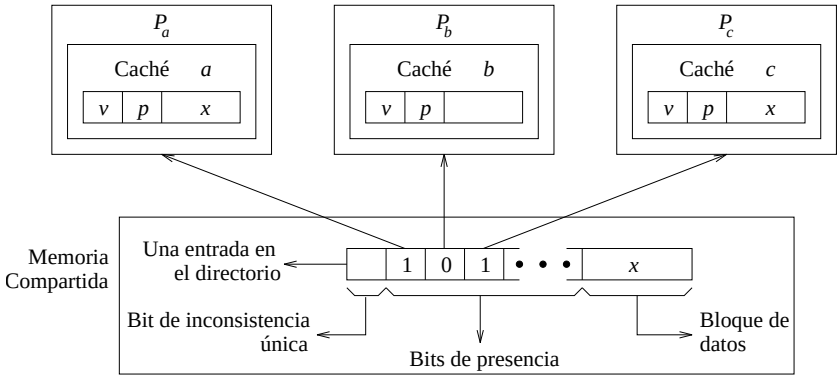


Figura 5.12: Directorio en el protocolo de mapeado completo.

Veamos qué acciones hay que realizar con este protocolo antes los fallos de escritura y lectura, así como los aciertos de escritura. Naturalmente no hay que hacer nada en especial frente a un acierto de lectura.

Fallo de lectura: Supongamos que la caché c envía una petición de fallo de lectura a la memoria. Si el bit de inconsistencia simple está activado, la memoria envía una petición de actualización a la caché que tenga el bit privado activo, o sea, a la única caché que tiene esa línea. La caché devuelve el último contenido del bloque a la memoria y desactiva su bit de privado. El bit de inconsistencia simple del bloque en el directorio central es también desactivado. La memoria activa el bit

de presencia correspondiente a la caché c y envía una copia del bloque a c. Una vez la caché c recibe la copia, activa el bit de valido y desactiva el de privado.
Si el bit de inconsistencia simple estaba desactivado, trae el bloque de la memoria activando el de válido y desactivando el de privado como en el caso anterior.

Fallo de escritura: Supongamos que la caché c envía una petición de fallo de escritura a la memoria. La memoria envía peticiones de invalidación a todas las cachés que tienen copias de ese bloque poniendo a cero sus bits de presencia. Las cachés aludidas invalidan la línea poniendo a cero el bit de línea válida enviado mensajes de reconocimiento a la memoria principal. Durante este proceso, si hay alguna caché (excepto la c) con una copia del bloque y con el bit de privado activado, la memoria se actualiza a sí misma con el contenido del bloque de esta caché. Una vez la memoria recibe todos los reconocimientos, activa el bit de presencia de c y manda la copia a la caché c. El bit de inconsistencia simple se pone a uno, puesto que sólo hay una copia. Una vez la caché recibe la copia, se modifica, y los bits de válido y privado se ponen a uno.

Acierto de escritura: Si el bit de privado es 0, c envía una petición de privacidad a la memoria. La memoria invalida todas las cachés que tienen copia del bloque (similar al caso de fallo de escritura), y luego pone el bit de inconsistencia simple a uno enviando un mensaje de reconocimiento a c. En el momento c recibe el reconocimiento de que las copias han sido invalidadas, modifica el bloque de caché y pone el bit de privado a uno.
Si el bit de privado era 1, entonces escribe sin más puesto que es la única caché con copia de esa línea.

El principal problema del mapeado completo es que el tamaño del directorio es muy grande. Tal y como hemos visto, este tamaño se puede calcular como:

$$\text{Tamaño del directorio} = (1 + \text{Número de cachés}) \times \text{Número de líneas de memoria}$$

Es decir, el tamaño del directorio es proporcional al número de procesadores multiplicado por el tamaño de la memoria. Como la memoria viene a ser habitualmente proporcional al número de procesadores (lo cual es estrictamente cierto en multiprocesadores con memoria compartida distribuida), resulta que el tamaño del directorio es directamente proporcional al cuadrado del número de procesadores, disparándose por tanto cuando el número de procesadores es elevado. Si además las líneas de caché son propiedad de sólo unos pocos procesadores, se ve que este esquema es muy ineficiente.

A continuación veremos una forma de aliviar estos problemas, y veremos también más adelante, en los protocolos de directorio distribuido, otras formas de reducir el directorio para que sea fácilmente escalable.

5.3.3. Protocolo de directorio limitado

Para evitar el crecimiento cuadrático del mapeado completo es posible restringir el número de copias de caché activas de forma simultánea de un bloque. De esta forma se limita el crecimiento del directorio hacia un valor constante.

Un protocolo general basado en directorio se puede clasificar usando la notación Dir_iX, donde Dir indica directorio, i es el número de punteros en el directorio, y X indica el número de copias que puede haber en el sistema que necesitan tener una referencia centralizada. En el caso de redes sin posibilidad de establecer protocolos de

sondeo entre procesadores o grupos de procesadores, la X vale NB donde N es el número de procesadores y B el número de bloques de la memoria de cada procesador.

Con esta nomenclatura, el protocolo basado en directorio de mapeado completo se representaría como $Dir_N NB$. En cambio, un protocolo basado en directorio limitado, donde $i < N$, se representaría como $Dir_i NB$.

El protocolo de coherencia para el protocolo de directorio limitado es exactamente igual que el visto para el mapeado completo, con la excepción del caso en que el número de peticiones de una línea en particular sea mayor que i tal y como se explica a continuación.

Supongamos que tenemos el directorio limitado a tan solo 2 punteros, que es lo que vendría a sustituir a los bits de presencia. Supongamos que ambos punteros están ocupados apuntando a copias válidas en dos cachés diferentes. Supongamos ahora que una tercera caché quiere leer en esa misma línea. Como no hay más punteros disponibles se produce lo que se llama un *desalojo* (*eviction*), que consiste en que la memoria invalida una de las cachés que utilizaban un puntero asignándole ese puntero que queda libre a la nueva caché.

Normalmente los procesadores suelen desarrollar su actividad local en zonas de memoria separadas de otros procesadores, eso quiere decir que con unos pocos punteros, incluso uno o dos, se puede mantener la actividad en el sistema sin que se produzcan *desalojos* y sin aumentar por tanto las latencias.

Los punteros son codificaciones binarias para identificar a cada procesador, por lo que en un sistema con N procesadores, son necesarios $\log_2 N$ bits para cada puntero. Con esto, el tamaño del directorio será:

$$\text{Tamaño del directorio} = (1 + \text{Punteros} \times \log_2 N) \times \text{Bloques de memoria}$$

Con las mismas suposiciones que para el mapeado completo, es fácil comprobar que el crecimiento del directorio es proporcional a $N \log_2 N$ que es sensiblemente inferior al de N^2 del protocolo con mapeado completo.

Este protocolo se considera *escalable*, con respecto a la sobrecarga de memoria, puesto que los recursos necesarios para su implementación crecen aproximadamente de forma lineal con el número de procesadores en el sistema.

5.3.4. Protocolo de directorio encadenado

Los tipos vistos anteriormente se aplican sobre todo a sistemas con directorio centralizado. Si el directorio se encuentra distribuido, lo que significa que la información del directorio se encuentra distribuida entre las memorias y/o cachés, se reduce el tamaño del directorio así como el cuello de botella que supone un directorio centralizado en multiprocesadores grandes. Directorios distribuidos hay de dos tipos, *protocolos de directorio jerárquico*, donde dividen el directorio entre varios *grupos* de procesadores, y *protocolos de directorio encadenado* basados en listas encadenadas de cachés.

Los protocolos de directorio jerárquico se utilizan a menudo en arquitecturas que consisten en un conjunto de grupos de procesadores *clusters* conectados por algún tipo de red. Cada cluster contiene un conjunto de procesadores y un directorio conectado a ellos. Una petición que no puede ser servida por las cachés en un directorio, es enviada a otros clusters tal y como determina el directorio.

Los protocolos de directorio encadenado mantienen una lista enlazada por punteros entre las cachés que tienen una copia de un bloque. La lista se puede recorrer en un sentido o en los dos dependiendo de que tenga uno o dos punteros cada nodo. Una entrada en el directorio apunta a una caché con una copia del bloque correspondiente a esa entrada; esta caché a su vez tiene un puntero que apunta a otra caché que tiene también la copia, y así sucesivamente. Por lo tanto, una entrada en el directorio contiene únicamente un único puntero que apunta a la cabeza de la lista. Con esto, el tamaño del directorio es proporcional al número de líneas de memoria y al logaritmo en base 2 del número de procesadores tal y como ocurría en el protocolo de directorio limitado. Naturalmente las cachés deben incluir también uno o dos punteros, dependiendo del tipo de lista, además de los bits propios de estado.

Un protocolo interesante, basado en directorio encadenado, es el que presenta el estándar *IEEE Scalable Coherent Interface* o *SCI*. Este interface estándar es escalable permitiendo la conexión de hasta 64K procesadores, memorias, o nodos de entrada/salida. Hay un puntero, llamado *puntero de cabeza*, asociado a cada bloque de memoria. El puntero de cabeza apunta a la primera caché en la lista. Además, se asignan a cada caché dos punteros, uno con el predecesor en la lista y otro con el sucesor. Se trata por tanto de una lista con doble enlace. La figura 5.13 muestra la estructura de memoria de un sistema SCI con los punteros de cabeza y los asociados a cada caché.

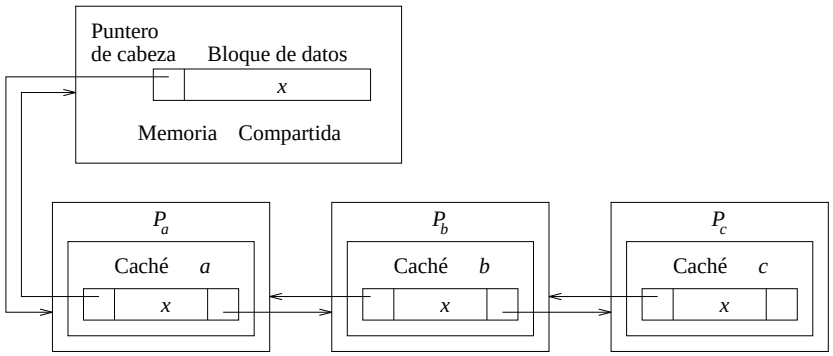


Figura 5.13: Protocolo de directorio encadenado en el estándar SCI.

Veamos las acciones a realizar en caso de fallos, aciertos, etc.

Fallo de lectura: Supongamos que la caché c envía una petición de fallo de lectura a la memoria. Si el bloque está en un estado *no cacheado*, es decir, no hay ningún puntero a ninguna caché, entonces la memoria manda el bloque a la caché. El estado del bloque se cambia a *cacheado* y el puntero de cabeza apunta a la caché. Si el estado del bloque es *cacheado*, es decir, el puntero de cabeza apunta a una caché con una copia válida del bloque, entonces la memoria envía el contenido del puntero, por ejemplo a, a c y actualiza el puntero de cabeza para que ahora apunte a c. La caché c entonces pone el puntero predecesor para que apunte al bloque de memoria. El siguiente paso consiste en que la caché c manda una petición a la caché a. Cuando a recibe la petición pone su puntero predecesor para que apunte a c y manda la línea de caché pedida a c.

Fallo de escritura: Supongamos que la caché c manda una petición de fallo de escritura a la memoria. La memoria entonces manda el puntero de cabeza (por ejemplo a) y actualiza el puntero de cabeza para apuntar a c. En este punto, la caché c que es el cabeza de lista, tiene autoridad para invalidar el resto de copias de caché para mantener sólo la copia que va a modificar. Esta invalidación se realiza enviando

la caché c una petición de invalidación a a; entonces a invalida su copia y manda su puntero sucesor (que por ejemplo apunta a b) a c. En este momento la caché c usa este puntero sucesor para enviar una petición de invalidación de línea a b. Estas operaciones continúan hasta que se invalidan todas las copias. Por último se realiza la escritura a la copia que es la única en la lista.

Acierto de escritura: Si la caché que escribe, c, es la única en la lista, simplemente actualiza la copia y no hace más.
Si la caché c es la cabeza de lista, entonces invalida todas las copias de caché para ser la única caché con copia. El proceso de invalidación es igual que el visto en el punto anterior. Por último escribe en la copia suya.
Si la caché c no está en cabeza entonces se quita a sí misma de la lista. Posteriormente interroga a la memoria para ver quién es la cabeza de lista y pone su puntero sucesor apuntando a la caché cabeza de lista en ese momento. La memoria pone su puntero para apuntar a c que será la nueva cabeza de lista. Ahora tenemos el caso anterior, así que se hace lo mismo, es decir, se invalidan el resto de copias y se escribe en la caché. Hay que hacer notar que en cualquier caso la lista se queda con un único elemento.

5.3.5. Rendimiento de los protocolos basados en directorio

Comparando los protocolos de directorio vistos hasta ahora se puede decir que los protocolos de mapeado completo ofrecen una mayor utilización del procesador que los encadenados y estos más que los de directorio limitado. Sin embargo, un directorio completamente mapeado requiere mucha más memoria que los otros dos. Por otro lado, la complejidad del directorio encadenado es mayor que la del limitado.

Comparando los protocolos basados en directorio con los de sondeo, podemos decir que los protocolos de directorio tienen la ventaja de ser capaces de restringir las peticiones de lectura/escritura sólo a aquellas cachés con una copia válida del bloque. Sin embargo, estos protocolos aumentan el tamaño de la memoria y las cachés debido a los bits y punteros extra. Los protocolos de sondeo tienen la ventaja de que su implementación es menos compleja. Sin embargo, estos protocolos de sondeo no se pueden escalar a un número elevado de procesadores y requieren cachés de doble puerto de alto rendimiento para permitir que el procesador ejecute instrucciones mientras sondea o fisga en el bus para ver las transacciones de las otras cachés.

El rendimiento de una máquina basada en directorio depende de muchos de los mismos factores que influían en el rendimiento de las máquinas basadas en bus (tamaño de la caché, número de procesadores y tamaño del bloque), como de la distribución de los fallos en la jerarquía de la memoria. La posición de los datos demandados depende tanto de la localización inicial como de los patrones de compartición. Empezaremos examinando el rendimiento básico de la caché en la carga de programas paralelos que utilizamos en la sección 5.2.7, para después determinar los efectos de los diferentes tipos de fallos.

Dado que las máquinas son más grandes y las latencias mayores que en el caso de multiprocesadores basados en snooping, empezaremos con una caché mayor (128 KB) y un bloque de 64 bytes. En las arquitecturas de memoria distribuida, la distribución de las peticiones entre locales y remotas es el factor clave para el rendimiento, ya que ésta afecta tanto al consumo del ancho de banda global como a la latencia vista por el nodo que realiza la petición. Por tanto, separaremos en las figuras de esta sección los

fallos de caché en locales y remotos. Al mirar estas figuras hay que tener en cuenta que, para estas aplicaciones, la mayoría de los fallos remotos se producen debido a fallos de coherencia, aunque algunos fallos de capacidad también puede ser remotos, y en algunas aplicaciones con una distribución de datos pobre, estos fallos pueden ser significativos.

Como muestra la figura 5.14, la tasa de fallos con estas cachés no se ve afectada por el cambio en el número de procesadores, con la excepción de Ocean, donde la tasa de fallos aumenta con 64 procesadores. Este aumento se debe al aumento de conflictos en el mapeo de la caché que ocurren cuando se reduce el tamaño de la malla, dando lugar a un aumento de los fallos locales, y al aumento de los fallos de coherencia, que son todos remotos.

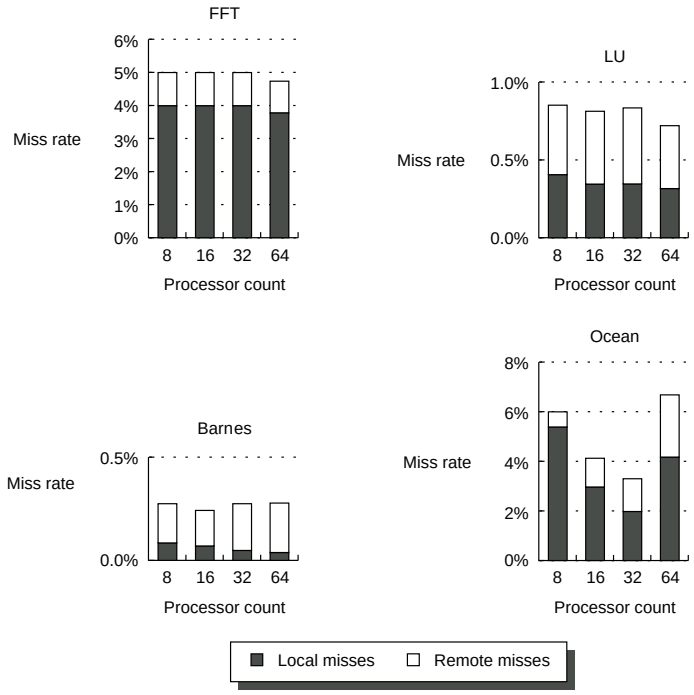


Figura 5.14: Tasa de fallo en función del número de procesadores.

La figura 5.15 muestra cómo cambia la tasa de fallos al incrementarse el tamaño de la caché, suponiendo una ejecución con 64 procesadores y bloques de 64 bytes. Estas tasas de fallo descienden tal como cabría esperar, aunque el efecto amortiguador causado por la casi nula reducción de los fallos de coherencia dan lugar a un menor decrecimiento de los fallos remotos que de los fallos locales. Para una caché de 512 KB, la tasa de fallo remota es igual o mayor que la tasa de fallo local. Tamaños de caché mayores sólo darán lugar a amplificar este fenómeno.

Finalmente, la figura 5.16 muestra el efecto de cambiar el tamaño del bloque. Dado que estas aplicaciones tienen buena localidad espacial, el incrementar el tamaño del bloque reduce la latencia de fallo, incluso para bloques grandes, aunque los beneficios en el rendimiento de utilizar los bloques de mayor tamaño son pequeños. Además, la mayoría parte de la mejora en la tasa de fallos se debe a los fallos locales.

Más que mostrar el tráfico de la memoria, la figura 5.17 muestra el número de bytes necesarios por referencia de datos en función del tamaño del bloque, dividiendo los

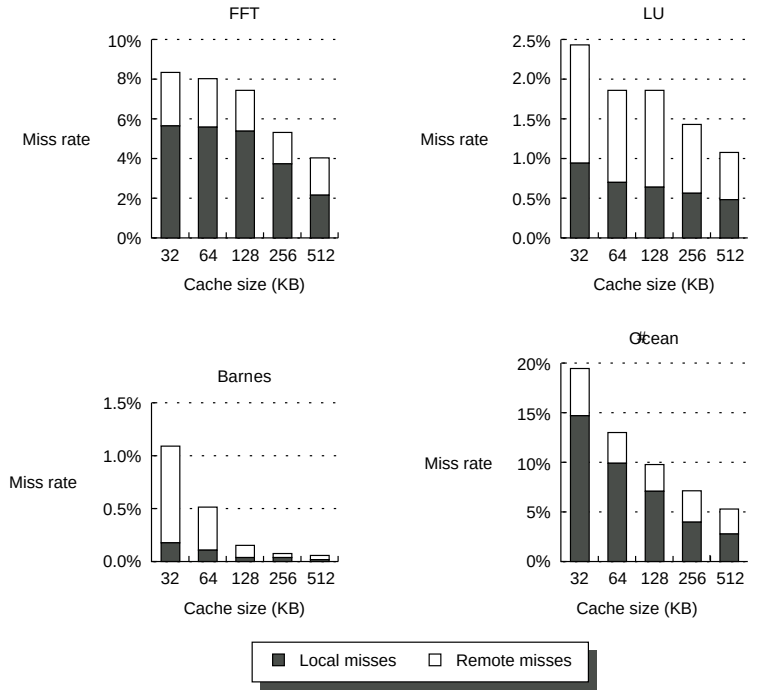


Figura 5.15: Tasa de fallo en función del tamaño de la caché.

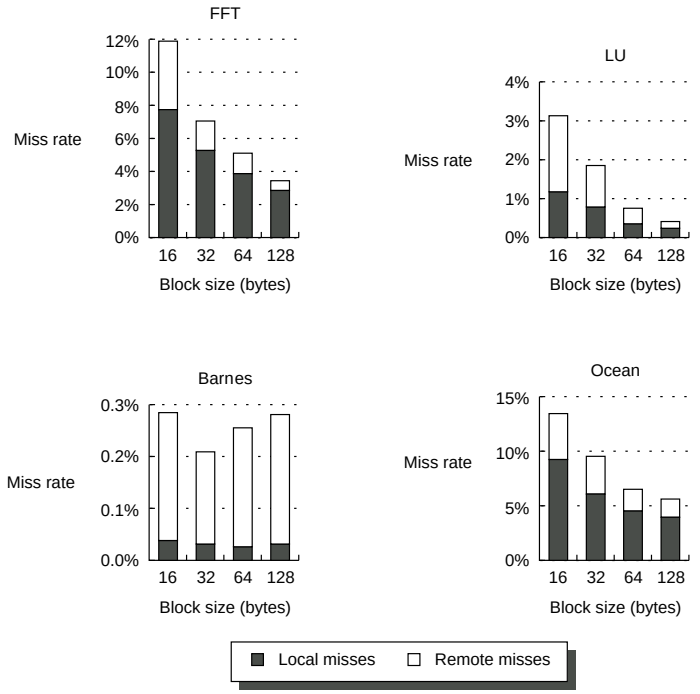


Figura 5.16: Tasa de fallo en función del tamaño del bloque suponiendo una caché de 128 KB y 64 procesadores.

requisitos de ancho de banda en local y global. En el caso de un bus, podemos unir ambos factores para determinar la demanda total para el bus y el ancho de banda de la memoria. Para un red de interconexión escalable, podemos usar los datos de la figura 5.17 para determinar el ancho de banda global por nodo y estimar ancho de banda de la bisección, como muestra el siguiente ejemplo.

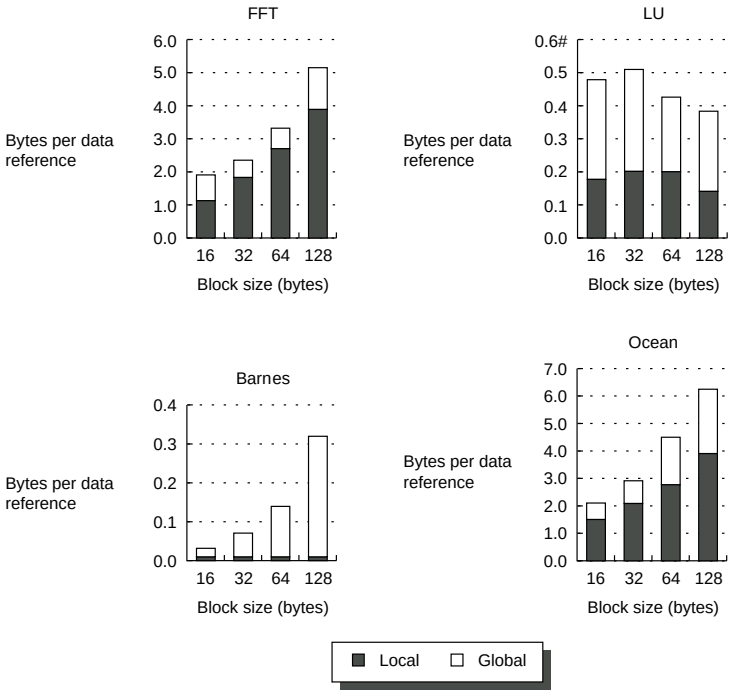


Figura 5.17: Número de bytes por referencia en función del tamaño del bloque.

Ejemplo Supongamos un multiprocesador de 64 procesadores a 200 MHz con una referencia a memoria por ciclo de reloj. Para un tamaño de bloque de 64 bytes, la tasa de fallo remota es del 0.7 %. Determinar el ancho de banda por nodo y el ancho de banda estimado para la bisección para FFT. Suponer que el procesador no se para ante peticiones a memoria remota; esto puede considerarse cierto si, por ejemplo, todos los datos remotos son precargados.

Respuesta El ancho de banda por nodo es simplemente el número de bytes por referencia multiplicado por la tasa de referencias: $0{,}007 \times 200 \times 64 = 90\ MB/sec$. Esta tasa es aproximadamente la mitad del ancho de banda del más rápido MPP escalable disponible en 1995. La demanda de ancho de banda por nodo excede a las redes de interconexión ATM más rápida existente en 1995 por un factor de 5, y excederá ligeramente a la siguiente generación de ATM

FFT realiza comunicaciones del tipo todos a todos, así que el ancho de banda de la bisección será igual a 32 veces el ancho de banda por nodo, o 2880 MB/sec. Para una máquina de 64 procesadores conectada mediante una malla 2D, el ancho de banda de la bisección crece proporcionalmente a la raíz cuadrada del número de procesadores. Así, para 64 procesadores el ancho de banda de la bisección es 8 veces el ancho de banda del nodo. En 1995, las interconexiones del tipo MPP ofrecen cerca de 200 MB/sec por nodo para un total de 1600 MB/sec. Para 64 nodos una conexión del tipo malla 3D dobla este ancho de banda de la bisección

($3200\,MB/sec$), que excede el ancho de banda necesario. La siguiente generación de mallas 2D también conseguirá alcanzar el ancho de banda necesario.

El ejemplo previo muestra la demanda del ancho de banda. Otro factor clave para los programas paralelos es el tiempo de acceso a memoria remota, o latencia. Para examinar este factor usaremos un ejemplo sobre una máquina basada en directorio. La figura 5.18 muestra los parámetros que supondremos para nuestra máquina. Supondremos que el tiempo de acceso para la primera palabra es de 25 ciclos y que la conexión a la memoria local tiene un ancho de 8 bytes, mientras que la red de interconexión tiene una anchura de 2 bytes. Este modelo ignora el efecto de la contención, que probablemente no es muy serio en los programas paralelos examinados, con la posible excepción de FFT, que usa comunicaciones del tipo todos a todos. La contención podría tener un serio impacto en el rendimiento para otro tipo de cargas de trabajo.

Characteristic	**Number of processor clock cycles**
Cache hit	1
Cache miss to local memory	$25 + \frac{\text{block size in bytes}}{8}$
Cache miss to remote home directory	$75 + \frac{\text{block size in bytes}}{2}$
Cache miss to remotely cached data (3-hop miss)	$100 + \frac{\text{block size in bytes}}{2}$

Figura 5.18: Características de la máquina basada en directorio del ejemplo.

La figura 5.19 muestra el coste en ciclos para una referencia a memoria, suponiendo los parámetros de la figura 5.18. Sólo se contabilizan las latencias de cada tipo de referencia. Cada barra indica la contribución de los éxitos de caché, fallos locales, fallos remotos y fallos remotos a una distancia de tres saltos. El coste está influenciado por la frecuencia total de los fallos de caché y actualizaciones, al igual que por la distribución de las localizaciones donde se satisface el fallo. El coste de una referencia a memoria remota permanece estable al aumentar el número de procesadores, excepto para Ocean. El incremento en la tasa de fallo en Ocean para 64 procesadores está claro a partir de la figura 5.14. Al aumentar la tasa de fallo, cabría esperar que el tiempo gastado en las referencias a memoria se incremente también.

Aunque la figura 5.19 muestra el coste de acceso a la memoria, que es el coste dominante en estos programas, un modelo completo del rendimiento debería considerar el efecto de la contención en el sistema de memoria, al igual que las pérdidas ocasionadas por los retrasos en la sincronización.

5.4. Modelos de consistencia de memoria

Este apartado se encuentra principalmente en [Tan95] y [CSG99], pero se puede ampliar el tema en [CDK96], [Hwa93] y [HP96].

La coherencia es esencial si se quiere transmitir información entre procesadores mediante la escritura de uno en una localización que leerá el segundo. Sin embargo,

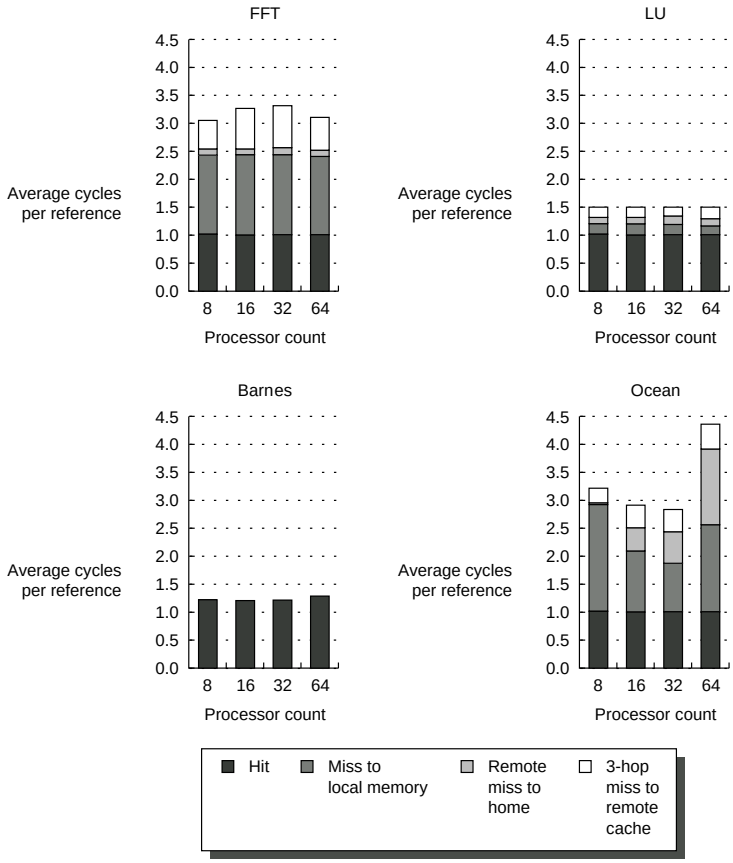


Figura 5.19: La latencia efectiva de las referencias a memoria en una máquina DSM depende de la frecuencia relativa de los fallos de caché y de la localización de la memoria desde donde se producen los accesos.

no se dice nada acerca de cuando el valor escrito debe hacerse visible. A menudo, en la escritura de un programa paralelo queremos asegurarnos que una lectura devuelve el valor de una determinada escritura. Es decir, queremos establecer un orden entre una escritura y una lectura. Normalmente, usamos alguna forma de evento de sincronización para asegurarnos esta dependencia.

Consideremos, por ejemplo, los fragmentos de código que se muestran en la figura 5.20 y que son ejecutados por los procesadores P_1 y P_2. De este código se deduce que el programador quiere que el proceso que se ejecuta en P_2 realice una espera activa hasta que el valor de la variable compartida *flag* cambie a 1, y después imprimir el valor de la variable A que debería ser 1, dado que dicho valor ha sido actualizado antes del valor de *flag* por el procesador P_1. En este caso, estamos accediendo a otra localización (*flag*) para preservar el orden entre los diferentes accesos a la misma localización (A). En particular, estamos suponiendo que la escritura de A es visible para el procesador P_2 antes que la escritura a *flag*, y que la lectura de flag por P_2 que hace que salgamos del bucle *while* se ha completado antes que la lectura de A. Esta ordenación entre los accesos de P_1 y P_2 no está garantizada por la coherencia, que, por ejemplo, únicamente requiere que el nuevo valor de A se haga visible al procesador P_2, no necesariamente antes de que el nuevo valor de *flag* sea observado.

Claramente, estamos esperando más del sistema de memoria que devolver el "último

```
        /* El valor inicial de A y flag es 0 */
  P1          P2
A=1       while (flag == 0); /* bucle vacío */
flag=1    print A
```

Figura 5.20: Requisitos de un evento de sincronización mediante flags.

valor escrito" para cada localización: Establecer un orden entre los accesos realizados por diferentes procesos a la localización (digamos A), en algunos casos esperamos que un sistema de memoria respete el orden de las lecturas y escrituras a *diferentes* localizaciones (A y *flag*) realizadas por un proceso dado. La coherencia no dice nada acerca del orden en el cual las escrituras realizadas por P_1 deben hacerse visibles a P_2, dado que estas operaciones se realizan sobre diferentes localizaciones. De manera similar, no dice nada acerca del orden en el cual las lecturas realizadas a diferentes localizaciones por P_2 se realizan de forma relativa a P_1. Por lo tanto, la coherencia por si sola no evita un resultado de 0 en el ejemplo. Claramente, necesitamos algo más que la coherencia para dar a un espacio de direcciones compartido una semántica clara, es decir, un modelo de ordenación que los programadores puedan usar para razonar acerca de los posibles resultados y por tanto de la corrección de sus programas.

Un *modelo de consistencia de la memoria* para un espacio de direcciones compartido especifica las restricciones en el orden en el cual las operaciones de memoria deben parecer haberse realizado (es decir, hacerse visibles a los procesadores) entre ellas. Esto incluye operaciones a las mismas localizaciones o diferentes localizaciones, y por el mismo proceso o por diferentes procesos. Por lo tanto, la consistencia de la memoria incluye la coherencia.

5.4.1. Consistencia secuencial

Antes de ver el modelo de consistencia secuencial, que es sin duda uno de los más implementados, se introduce el problema de la consistencia de la memoria con un modelo ideal que no es implementable en la práctica, pero que es lo que probablemente uno esperaría del comportamiento de un sistema multiprocesador cualquiera.

Modelo de consistencia estricta

Este modelo es el caso ideal en el cual cualquier escritura se ve instantáneamente por cualquier lectura posterior. Dicho de otra manera más formal: "Cualquier lectura de cierta posición de memoria X devuelve el valor almacenado por la escritura más reciente realizada sobre X".

Hay que hacer notar que **este modelo de consistencia no se puede implementar en ningún sistema real**. Efectivamente, como la distancia entre los procesadores y la memoria no es nula, la señal que viaja entre el procesador que escribe y la memoria sufre un cierto retraso. Si durante este retraso, en el cual la memoria no tiene aún el nuevo dato escrito, otro procesador decide leer, no va a leer el valor que se acaba de mandar a la memoria, que aún no ha llegado, sino el que hubiera anteriormente.

Por lo tanto, la única forma en la que se podría implementar este modelo es suponiendo que los datos viajan de forma instantánea (velocidad infinita), lo cual va en contra de la teoría general de relatividad.

El modelo de consistencia secuencial

Cuando se discutieron las características de diseño fundamentales de una arquitectura de comunicación, se describió de forma informal un modelo de ordenación deseable para un espacio de direcciones compartido: el que nos asegura que el resultado de un programa multihilo (*multithread*) ejecutándose con cualquier solapamiento de los distintos hilos sobre un sistema monoprocesador sigue siendo el mismo en el caso de que algunos de los hilos se ejecuten en paralelo sobre diferentes procesadores. La ordenación de los accesos a los datos dentro de un proceso es la del programa, y entre los procesos viene dada por alguno de los posibles solapamientos entre los ordenes de los diferentes programas. Así, en el caso del multiprocesador, éste no debe ser capaz de hacer que los resultados sean visibles en el espacio de direcciones compartido en un forma diferente a la que se pueda generar por los accesos solapados de diferentes procesos. Este modelo intuitivo fue formalizado por Lamport en 1979 como *consistencia secuencial*, que se define como sigue:

> *Consistencia secuencial.* Un multiprocesador es secuencialmente consistente si el resultado de cualquier ejecución es la misma que si las operaciones de todos los procesadores se realizaran en algún orden secuencial, y las operaciones de cada procesador individual ocurren en esta secuencia en el orden especificado por el programa.

La figura 5.21 representa el modelo de abstracción proporcionado a los programadores por un sistema de consistencia secuencial. Varios procesadores *parecen* compartir una *única* memoria lógica, incluso aunque en la máquina real la memoria principal pueda estar distribuida entre varios procesadores, incluso con sus cachés y buffers de escritura privados. Cada procesador parece emitir y completar las operaciones a la memoria de una vez y de forma atómica en el orden del programa —es decir, una operación a la memoria no parece haber sido emitida hasta que la previa ha sido completada— y la memoria común parece servir a estas peticiones una cada vez, de manera entrelazada de acuerdo con un orden arbitrario. Las operaciones a la memoria aparecen como *atómicas* dentro de este orden; es decir, deben aparecer globalmente (a todos los procesadores) como si una operación en este orden entrelazado consistente se ejecuta y se completa antes de que empiece la siguiente.

Al igual que en el caso de la coherencia, no es importante en qué orden se emitan realmente las operaciones a la memoria o incluso cuándo se completan. Lo que importa es que parezcan completarse en un orden que no viole la consistencia secuencial. Veámoslo mediante el ejemplo que se muestra en la figura 5.22.

En este ejemplo, el resultado (0,2) para (A,B) no estaría permitido bajo la consistencia secuencial (preservando nuestra intuición) dado que parecería que las escrituras de A y B en el proceso P_1 se han ejecutado fuera de orden. Sin embargo, las operaciones de memoria podrían ejecutarse y completarse en el orden 1b, 1a, 2b, 2a. No importa que se hayan completado fuera de orden, dado que el resultado (1,2) de la ejecución es la

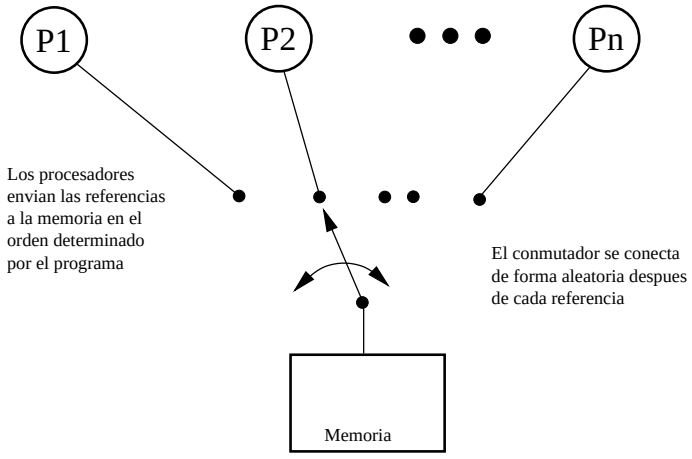


Figura 5.21: Abstracción desde el punto de vista del programador del subsistema de memoria bajo el modelo de consistencia secuencial.

P_1	P_2
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

Figura 5.22: Órdenes entre los accesos sin sincronización.

misma que si las operaciones se hubiesen ejecutado y completado en el orden del programa. Por otra parte, la ejecución 1b, 2a, 2b, 1a no cumpliría la consistencia secuencial, ya que produciría el resultado (0,2) que no está permitido bajo este paradigma.

La implementación de la consistencia secuencial hace necesario que el sistema (software y hardware) preserve las restricciones intuitivas definidas arriba. Existen dos restricciones. La primera es que las operaciones de memoria de un proceso se hagan visibles en el *orden del programa*. El segundo requisito, necesario para garantizar que el orden total sea consistente para todos los procesos, es que las operaciones aparezcan atómicas; es decir, que parezca que una operación se ha completado con respecto a todos los procesos antes que la siguiente operación en el orden total haya sido emitida. El truco de la segunda restricción es hacer que las escrituras parezcan atómicas, especialmente en un sistema con varias copias de un bloque que necesita ser informado ante una escritura. La *atomicidad de las escrituras*, incluidas en la definición de consistencia secuencial, implica que la posición en la cual parece realizarse una escritura en el orden total debe ser la misma con respecto a todos los procesadores. Esto asegura que nada de lo que un procesador haga *después* de que haya visto el nuevo valor producido por una escritura se hace visible a los otros procesadores antes de que ellos también hayan visto el nuevo valor para esa escritura. De hecho, mientras que la coherencia (serialización de las escrituras) dice que las escrituras a la misma localización deben parecerles a todos los procesadores que han ocurrido en el mismo orden, la consistencia secuencial dice que todas las escrituras (a cualquier localización) deben parecerles a todos los procesadores que han ocurrido en el mismo orden. El siguiente ejemplo muestra por qué es importante la escritura atómica.

Ejemplo Consideremos los tres procesos que se muestran en la figura 5.23. Mostrar

cómo la no preservación de la atomicidad en la escritura viola la consistencia secuencial.

Respuesta Dado que P_2 espera hasta que A sea 1 y después hace que B valga 1, y dado que P_3 espera hasta que B vale 1 y únicamente después lee el valor de A, por transitividad podríamos deducir que P_3 encontrará que A vale 1. Si P_2 puede continuar después de la lectura de A y escribir B antes de tener garantizado que P_3 ha visto el nuevo valor de A, entonces P_3 puede leer el nuevo valor de B y el viejo valor de A de su caché, violando nuestra intuición de la consistencia secuencial.

P_1		P_2		P_3
A=1;	$\longrightarrow$	while (A==0);		
		B=1;	$\longrightarrow$	while (B==0);
				print A

Figura 5.23: Ejemplo que ilustra la importancia de la atomicidad de las escrituras para la consistencia secuencial.

El orden del programa de cada proceso impone un orden parcial sobre el conjunto de todas las operaciones; es decir, impone una ordenación sobre el conjunto de operaciones que emite dicho proceso. Un intercalamiento de las operaciones de diferentes procesos definen un orden total en el conjunto de todas las operaciones. En realidad, para cada posible intercalamiento se define un orden total diferente dando lugar a un gran número de posibles órdenes totales. Este hecho da lugar a las siguientes definiciones:

Ejecución secuencialmente consistente. Una ejecución de un programa se dice que es secuencialmente consistente si los resultados que produce son los mismos que los que produce uno de los posibles órdenes totales (intercalados como se ha definido anteriormente). Es decir, debe existir un orden total o intercalación de órdenes de los programas de los procesos que den lugar al mismo resultado.

Sistema secuencialmente consistente. Un sistema es secuencialmente consistente si cualquier posible ejecución de ese sistema se corresponde con (produce el mismo resultado que) alguno de los posibles ordenes totales definidos arriba.

Condiciones suficientes para preservar la consistencia secuencial

Es posible definir un conjunto de condiciones suficientes que garanticen la consistencia secuencial en un multiprocesador. El siguiente conjunto se usa de forma habitual debido a su relativa simplicidad sin ser excesivamente restrictivas:

1. Todo proceso emite las peticiones a la memoria en el orden especificado por el programa.
2. Después de emitir una operación de escritura, el proceso que la emitió espera ha que se haya realizado la escritura antes de emitir su siguiente operación.
3. Después de emitir una operación de lectura, el proceso que la emitió espera a que la lectura se haya completado, *y* también espera a que se complete la escritura que generó el valor devuelto por la lectura, antes de emitir su siguiente operación. Es

decir, si la escritura que generó el valor que se obtiene mediante la lectura ha sido realizada con respecto a este procesador (y debe haberlo hecho si se ha devuelto ese valor) entonces el procesador debe esperar hasta que la escritura se haya realizado con respecto a todos los procesadores.

La tercera condición es la que asegura la atomicidad de la escritura, y es bastante exigente. No es simplemente una restricción local, ya que la lectura debe esperar hasta que escritura que le precede de forma lógica sea visible de forma global. Obsérvese que estas son condiciones suficientes, más que condiciones necesarias. La consistencia secuencial puede preservarse con menos serialización en muchas situaciones.

5.4.2. Otros modelos de consistencia

Con el orden del programa definido en términos del programa fuente, está claro que para que se cumplan estas condiciones el compilador no debe cambiar el orden de las operaciones a la memoria en su presentación al procesador. Desafortunadamente, muchas de las optimizaciones empleadas en los compiladores y procesadores violan esta condición. Por ejemplo, los compiladores reordenan rutinariamente los accesos a diferentes localizaciones dentro de un proceso, así el procesador puede emitir accesos a memoria sin cumplir el orden del programa visto por el programador. Explicitamente, los programas paralelos usan compiladores uniprocesador, que sólo se preocupan de preservar las dependencias a la misma localización.

Incluso en el caso de que el compilador preserve el orden del programa, los modernos procesadores usan mecanismos sofisticados como buffers de escritura, memoria entrelazada, encauzamiento y técnicas de ejecución desordenada. Éstas permiten que las operaciones a la memoria de un proceso sean emitidas, ejecutadas y/o completadas sin seguir el orden del programa. Estas optimizaciones en la arquitectura y el compilador funcionan en los programas secuenciales, ya que en ellos sólo es necesario preservar las dependencias entre los accesos a la misma localización de la memoria.

Debido a las fuertes restricciones que impone la consistencia secuencial, se han propuesto otros modelos más relajados que permiten implementaciones de más rendimiento y que todavía preservan un modelo de programación simple. De hecho, existen ciertos modelos relajados que mantienen la propiedad de una semántica de ejecución idéntica a la que sería bajo un modelo de consistencia secuencial. En la figura 5.24 se muestran algunos de los modelos propuestos. Una flecha de un modelo a otro indica que el segundo permite más optimizaciones que el primero y, por tanto, proporciona mayores prestaciones a costa, posiblemente, de un interfaz más complejo. Dos modelos no conectados con flechas no pueden compararse en términos de rendimiento o programabilidad. Las flechas discontinuas aparecen en el caso de considerar un revisión particular de *processor consistency* y *release consistency* propuesta por J. Hennessy, K. Gharachorloo y A. Gupta[1].

Para entender las diferencias entre los diferentes modelos relajados y las posibles implicaciones para una implementación de los mismos, la forma más sencilla es definir los modelos en términos de las ordenaciones entre lecturas y escrituras *realizadas por un único procesador* que preserva cada modelo. Es posible establecer cuatro ordenaciones en función de las cuatro posibles dependencias existentes entre dos instrucciones:

[1]Revision to "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocesors", Technical Repor CSL-TR-93-568, Stanford University, April 1993.

--

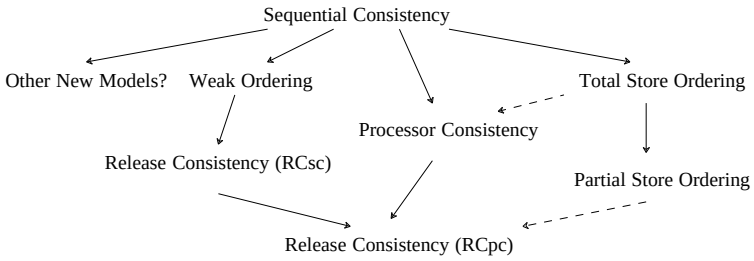


Figura 5.24: Modelos de consistencia de memoria.

1. $R \rightarrow R$: una lectura seguida de una lectura.
2. $R \rightarrow W$: una lectura seguida de una escritura, que siempre se preserva si las operaciones son sobre la misma dirección, dado que se trata de una antidependencia.
3. $W \rightarrow W$: una escritura seguida de otra escritura, que siempre se preserva si las operaciones son sobre la misma dirección, dado que se trata de una dependencia de salida.
4. $W \rightarrow R$: una escritura seguida de una lectura, que siempre se preserva si son sobre la misma dirección, dado que se trata de una dependencia verdadera.

Si existe una dependencia entre una lectura y una escritura, la semántica de un programa ejecutado sobre un uniprocesador exige que las operaciones estén ordenadas. Si no existen dependencias, el modelo de consistencia de la memoria determina qué órdenes deben preservarse. Un modelo de consistencia secuencial exige que se preserven los cuatro órdenes, lo que equivale a considerar un único módulo de memoria centralizado que serializa las operaciones de todos los procesadores, o asumir que todas las lecturas y escrituras a la memoria se comportan como barreras.

Cuando se relaja una ordenación, simplemente queremos decir que permitimos la finalización anticipada de una operación por parte de un procesador. Por ejemplo, relajar la ordenación $W \rightarrow R$ significa que permitimos que una lectura que era posterior a una escritura se complete antes. Hay que recordar que un escritura no se completa (realiza) hasta que se han completado todas las invalidaciones, así, permitiendo que ocurra una lectura después de un fallo en una operación de escritura anterior pero antes de que las invalidaciones se hayan realizado ya no se preserva este orden.

En realidad, un modelo de consistencia no restringe el orden de los eventos, sino que dice qué ordenaciones pueden ser *observadas*. Por ejemplo, en la consistencia secuencial, el sistema debe parecer que preserva los cuatro ordenes descritos anteriormente, aunque en la práctica se permite la reordenación. Esta sutileza permite implementaciones que usan trucos que reordenan eventos sin permitir que dichas reordenaciones sean observadas. En la consistencia secuencial una implementación puede, por ejemplo, permitir que un procesador, P, inicie otra escritura antes de que la escritura anterior haya sido completada, siempre que P no permita que el valor de la última escritura sea visible antes de que la escritura anterior se complete.

El modelo de consistencia también debe definir la ordenación existente entre los accesos a las variables de sincronización, que actúan como barreras, y el resto de los

accesos. Cuando una máquina implementa la consistencia secuencial, todas las lecturas y escrituras, incluyendo los accesos de sincronización, son barreras y por tanto deben mantenerse ordenadas. Para modelos más débiles, debemos especificar las restricciones en la ordenación impuestas por los accesos de sincronización, al igual que las restricciones sobre las variables ordinarias. La restricción en la ordenación más simple es que que todo acceso de sincronización se comporta como una barrera. Si utilizamos S para indicar un acceso a una variable de sincronización, también podremos indicar estas restricciones con la notación $S \rightarrow W$, $S \rightarrow R$, $W \rightarrow S$ y $R \rightarrow S$. Hay que recordar que un acceso de sincronización es también una lectura o una escritura y que está afectado por la ordenación respecto a otros accesos de sincronización, lo que significa que existe una ordenación implícita $S \rightarrow S$.

Modelo de consistencia PRAM o de procesador

El primer modelo que examinaremos relaja la ordenación entre una escritura y una lectura (a direcciones diferentes), eliminando la ordenación $W \rightarrow R$; este modelo fue usado por primera vez en la arquitectura IBM 370. Estos modelos permiten almacenar las escrituras en buffers que pueden ser sobrepasadas por las lecturas, lo que ocurre siempre que el procesador permita que una lectura proceda antes de garantizar que una escritura anterior de ese procesador ha sido vista por todos los procesadores. Este modelo permite a la máquina ocultar algo de la latencia de una operación de escritura. Además, relajando únicamente una ordenación, muchas aplicaciones, incluso aquellas que no están sincronizadas, funcionan correctamente, aunque será necesaria una operación de sincronización para asegurar que se complete una escritura antes de que se realice una lectura. Si se ejecuta una operación de sincronización antes de una lectura (por ejemplo, un patrón $W...S...R$), entonces las ordenaciones $W \rightarrow S$ y $S \rightarrow R$ nos aseguran que la escritura será realizada antes que la lectura. A este modelo se le ha denominado *Processor consistency* o *total store ordering* (TSO), y muchas máquinas han utilizado de forma implícita este método. Este modelo es equivalente a realizar las escrituras con barreras. En la figura 5.25 resumiremos todos los modelos, mostrando los ordenes impuestos, y mostraremos un ejemplo en la figura 5.26.

Ordenamiento por almacenamiento parcial

Si permitimos que las escrituras no conflictivas se completen de forma desordenada, relajando la ordenación $W \rightarrow W$, llegamos a un modelo que ha sido denominado como *partial store ordering* (PSO). Desde el punto de vista de la implementación, esto permite el encauzamiento o solapamiento de las operaciones de escritura.

Consistencia débil

Una tercera clase de modelos relajados eliminan las ordenaciones $R \rightarrow R$ y $R \rightarrow W$, además de las otras dos. Estos modelos, denominados *weak ordering*, no preservan el orden entre las referencias, a excepción de las siguientes:

- Una lectura o escritura se completará antes que cualquier operación de sincronización ejecutada por el procesador después de la lectura o escritura según el orden del programa.

- Una operación de sincronización se completará antes que cualquier lectura o escritura que ocurra según el orden del programa después de la operación

Como se muestra en la figura 5.25, las únicas ordenaciones impuestas por la *ordenación débil* son las creadas por las operaciones de sincronización. Aunque hemos eliminado los ordenes $R \rightarrow R$ y $R \rightarrow W$, el procesador sólo puede aprovecharse de este hecho si dispone de lecturas no bloqueantes. En caso contrario, el procesador implementa implícitamente estos dos órdenes, dado que no se pueden ejecutar más instrucciones hasta que se complete la lectura. Incluso con lecturas no bloqueantes, la ventaja del procesador puede verse limitada dado que la principal ventaja ocurre cuando una lectura produce un fallo de caché y es improbable que el procesador pueda mantenerse ocupado durante los centenares de ciclos que se necesita para manejar un fallo de caché. En general, la principal ventaja de todos los modelos de consistencia débil viene del ocultamiento de la latencia de escritura más que de las latencias de lectura.

Model	**Used in**	**Ordinary orderings**	**Synchronization orderings**
Sequential consistency	Most machines as an optional mode	R →R, R →W, W→R, W→W	S→W, S→R, R→S, W→S, S→S
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	R →R, R →W, W→W	S→W, S→R, R→S, W→S, S→S
Partial store order	SPARC	R →R, R →W	S→W, S→R, R→S, W→S, S→S
Weak ordering	PowerPC		S→W, S→R, R→S, W→S, S→S
Release consistency	Alpha, MIPS		S_A→W, S_A→R, R→S_R, W→S_R, S_A→S_A, S_A→S_R, S_R→S_A, S_R→S_R

Figura 5.25: Ordenaciones impuestas por varios modelos de consistencia. Se muestran tanto los accesos ordinarios como los accesos a variables de sincronización.

Modelo de consistencia de liberación (*release*)

Es posible obtener un modelo más relajado todavía extendiendo la ordenación débil. Este modelo, llamado *release consistency* (consistencia en la liberación), distingue entre operaciones de sincronización que se usan para *adquirir* el acceso a una variable compartida (denotadas por S_A) y aquellas que *liberan* un objeto para permitir que otro procesador puede adquirir el acceso (denotadas por S_R). La consistencia en la liberación se basa en la observación de que en los programas sincronizados es necesaria una operación de adquisición antes de usar un dato compartido, y que una operación de liberación debe seguir a todas las actualizaciones a los datos compartidos y preceder en el tiempo a la siguiente operación de adquisición. Esto nos permite relajar ligeramente la ordenación observando que una lectura o escritura que precede a una adquisición no necesitan completarse antes de la adquisición, y que una lectura o escritura que sigue a una liberación no necesitan esperar a dicha liberación. Por tanto, las ordenaciones que se preservan son las que implican únicamente a S_A y S_R, como se muestra en la figura 5.25; como muestra en el ejemplo de la figura 5.26, este modelo impone el menor número de ordenes de los cinco modelos.

Para comparar los modelos de *release consistency* y *weak ordering*, consideremos

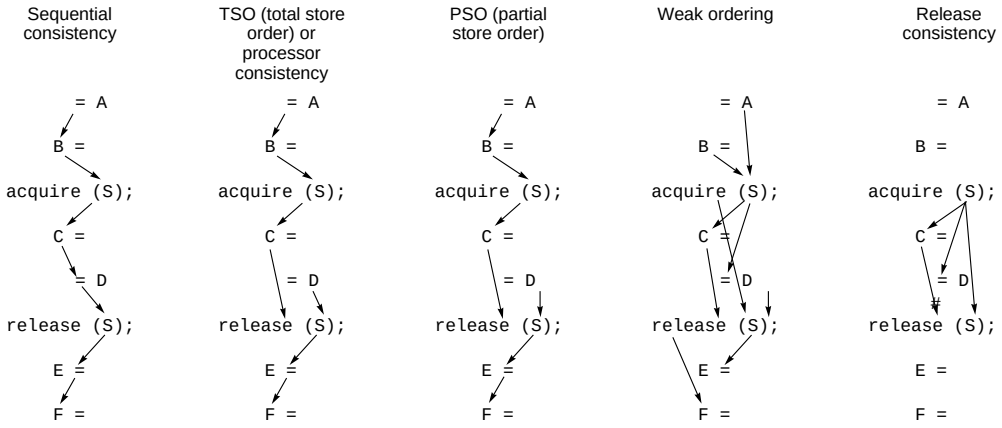


Figura 5.26: Ejemplos de los cinco modelos de consistencia vistos en esta sección que muestran la reducción en el número de órdenes impuestos conforme los modelos se hacen más relajados.

qué ordenaciones son necesarias para este último modelo, para ello descompondremos cada S en S_A y S_R. Esto nos llevaría a ocho ordenaciones en relación con los accesos sincronizados y los accesos ordinarios y cuatro ordenaciones en relación únicamente con los accesos sincronizados. Con esta descripción, podemos ver que cuatro de las ordenaciones impuesta en *weak ordering* no aparecen en el modelo de *release consistency*: $W \rightarrow S_A$, $R \rightarrow S_A$, $S_R \rightarrow R$ y $S_R \rightarrow W$.

El modelo de *release consistency* es uno de los menos restrictivos, permitiendo un fácil chequeo y asegurando el funcionamiento de los programas sincronizados. Mientras que la mayoría de las operaciones son una adquisición o una liberación (una adquisición normalmente lee una variable de sincronización y la actualiza de forma atómica, una liberación únicamente escribe en ella), algunas operaciones, como una barrera, actúa de adquisición y liberación a la vez en cuyo caso el modelo pasa a ser equivalente al modelo de *weak ordering*.

También es posible considerar ordenaciones más débiles. Por ejemplo, en el modelo de *release consistency* no asociamos localizaciones de memoria a variables de sincronización específicas. Si obligamos a la adquisición de la misma variable de sincronización, V, antes de acceder a una dirección de memoria particular, M, podemos relajar el orden de accesos a M y la adquisición y relajación de las variables de sincronización distintas de V.

Consistencia de liberación perezosa

Consistencia de ingreso

5.4.3. Implementación de los modelos relajados

Los modelos de consistencia relajados pueden implementarse mediante la utilización de un poco de hardware adicional. La mayor parte de la complejidad recae sobre la implementación de memorias o de los sistemas de interconexión que saquen provecho de un modelo relajado. Por ejemplo, si la memoria o la interconexión no permite la

múltiples peticiones por parte de un procesador, los beneficios de los modelos relajados más ambiciosos serán pequeños. Afortunadamente, la mayoría de los beneficios pueden conseguirse permitiendo un pequeño número de peticiones en curso de escritura y una de lectura por procesador. En esta sección describiremos implementaciones directas de los modelos *processor consistency* y *release consistency*.

El modelo TSO (*processor consistency*) se implementa permitiendo que un fallo de lectura sobrepase a las escrituras pendientes. Un buffer de escritura que soporte un chequeo para determinar si existen operaciones de escritura pendientes en el buffer a la misma dirección que una operación de lectura, junto con una memoria y un sistema de interconexión que soporte dos referencias pendientes por nodo, es suficiente para implementar este esquema. Cualitativamente, la ventaja de la consistencia a nivel de procesador sobre la consistencia secuencial es que permite ocultar la latencia de los fallos en las escrituras.

El modelo de *release consistency* permite una ocultación adicional de la latencia de escritura, y si el procesador soporta lecturas no bloqueantes, permite ocultar la latencia de las lecturas. Para permitir el ocultamiento de la latencia de las escrituras tanto como sea posible, el procesador debe permitir varias escrituras pendientes y que los fallos en las lecturas sobrepasen a las escrituras pendientes. Para maximizar el rendimiento, las escrituras deben completarse y eliminarse el buffer de escritura tan pronto como sea posible, lo que permite el avance de las lecturas pendientes. El soporte de la finalización temprana de las escrituras requiere que una escritura se complete tan pronto como los datos estén disponibles y antes de que se completen las invalidaciones pendientes (dado que nuestro modelo de consistencia lo permite). Para implementar este esquema, el procesador debe ser capaz de llevar la cuenta de las invalidaciones de cada escritura emitida. Después de que cada invalidación sea confirmada, el contador de invalidaciones pendientes para esa escritura se decrementa. Debemos asegurarnos que todas las invalidaciones pendientes de todas las escrituras emitidas se han completado antes de permitir que se complete una operación de liberación, de tal manera que sólo es necesario chequear los contadores de invalidaciones pendientes sobre cualquier escritura emitida cuando se ejecuta una liberación. La liberación se retiene hasta que todas las invalidaciones de todas las escrituras se han completado. En la práctica, se limita el número de escrituras pendientes en un momento dado, con lo que se facilita el seguimiento de las mismas y de las invalidaciones pendientes.

Para ocultar la latencia de las escrituras debemos disponer de una máquina que permita lecturas no bloqueantes; en caso contrario, cuando el procesador se bloquea, el progreso sería mínimo. Si las lecturas son no bloqueantes podemos simplemente permitir que se ejecuten, sabiendo que las dependencias entre los datos preservarán una ejecución correcta. Es poco probable, sin embargo, que la adición de lecturas no bloqueantes a un modelo de consistencia relajado mejore el rendimiento sustancialmente. Esta ganancia limitada se debe al hecho de que los tiempo de respuesta en un multiprocesador en el caso de fallo en una lectura sea grande y la habilidad de ocultar esta latencia por parte del procesador es limitada. Por ejemplo, si las lecturas son no bloqueantes pero se ejecutan en orden por parte del procesador, entonces éste se bloqueará después de algunos ciclos de forma casi segura. Si el procesador soporta lecturas no bloqueantes y ejecución desordenada, se bloqueará tan pronto como el buffer de reordenación o las estaciones de reserva se llenen. Es muy probable que este hecho ocurra en un centenar de ciclos, mientras que un fallo puede costar un millar de ciclos.

5.4.4. Rendimiento de los modelos relajados

El rendimiento potencial de un modelo de consistencia más relajado depende de las capacidades de la máquina y de la aplicación que se ejecute. Para examinar el rendimiento de un modelo de consistencia de la memoria, debemos definir en primer lugar las características del hardware:

- El cauce emite una instrucción por ciclo de reloj y el manejo puede ser estático o dinámico. La latencia de todas las unidades funcionales es de un ciclo.
- Un fallo de caché supone 50 ciclos de reloj.
- La CPU incluye un buffer de escritura con capacidad para 16 escrituras.
- Las cachés son de 64KB y el tamaño de las líneas es de 16 bytes.

Para dar una visión de los aspectos que afectan al rendimiento potencial con diferentes capacidades hardware, consideraremos cuatro modelos hardware:

1. SSBR (*statically scheduled with blocking reads*) El procesador realiza una ejecución estática y las lecturas que incurren en fallo en la caché bloquean al procesador.
2. SS (*statically scheduled*) El procesador realiza una ejecución estática pero las lecturas no causan el bloqueo del procesador hasta que se usa el resultado.
3. DS16 (*dynamically scheduled with a 16-entry reorder buffer*) El procesador permite una ejecución dinámica y tiene un buffer de reordenación que permite tener hasta 16 instrucciones pendientes de cualquier tipo, incluyendo 16 instrucciones de acceso a memoria.
4. DS64 (*dynamically scheduled with a 64-entry reorder buffer*) El procesador permite una ejecución dinámica y tiene un buffer de reordenación que permite hasta 64 instrucciones pendientes de cualquier tipo. Este buffer es, potencialmente, lo suficientemente grande para ocultar la totalidad de la latencia ante un fallo de lectura en la caché.

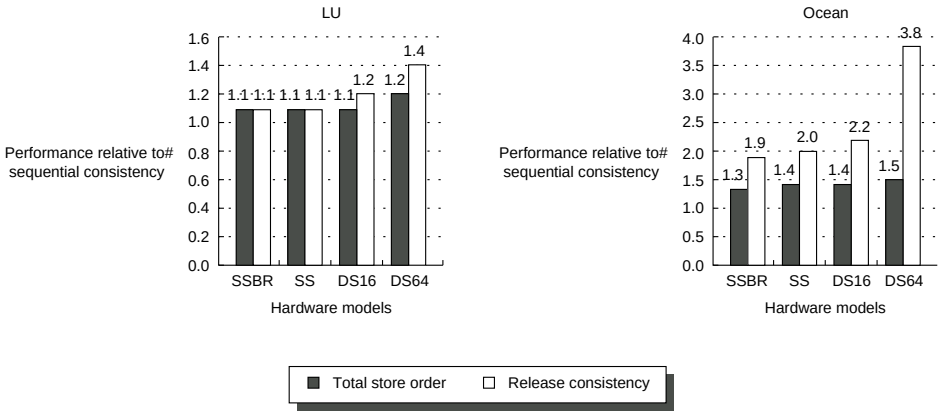


Figura 5.27: Rendimiento de los modelos de consistencia relajados sobre una variedad de mecanismos hardware.

La figura 5.27 muestra el rendimiento relativo para dos programas paralelos, LU y Ocean, para estos cuatro modelos hardware y para dos modelos de consistencia diferentes: TSO y *release consistency*. El rendimiento mostrado es relativo al rendimiento bajo

una implementación directa de la consistencia secuencial. El aumento de rendimiento observado es mucho mayor en el caso de Ocean, ya que este programa presenta un ratio de fallos de caché mucho mayor y tiene una fracción significativa de fallos de escritura. Al interpretar los datos de la figura 5.27 hay que recordar que las cachés son bastantes pequeñas. Muchos diseñadores habrían incrementado el tamaño de la caché antes de incluir lecturas no bloqueantes o empezar a pensar en ejecución dinámica de las instrucciones. Esto hubiera hecho descender el ratio de fallos y las ventajas de los modelos relajados, al menos sobre estas aplicaciones.

Para finalizar, decir que, en la actualidad, la mayoría de las máquinas que soportan alguna clase de modelo de consistencia relajado, variando desde la *consistencia a nivel de procesador* a la *consistencia en la liberación*, y casi todos soportan la consistencia secuencial como una opción. Dado que la sincronización depende mucho de la máquina y es causa de errores, las expectativas nos dicen que la mayoría de los programadores usarán librerías estándar de sincronización para escribir sus programas sincronizados, haciendo la elección de un modelo de consistencia relajado invisible al programador y permitiendo mayor rendimiento.

5.5. Sincronización

Los mecanismos de sincronización suelen implementarse mediante rutinas software que descansan en las instrucciones de sincronización proporcionadas por el hardware. Sin esta capacidad, el coste de construir las primitivas básicas de sincronización sería demasiado alto y se incrementaría al incrementarse el número de procesadores. Estas primitivas forman los bloques de construcción básicos para implementar una amplia variedad de operaciones de sincronización a nivel de usuario, incluyendo elementos tales como los cerrojos y las barreras. Durante años ha habido un considerable debate sobre qué primitivas hardware deben proporcionar las máquinas multiprocesador para implementar estas operaciones de sincronización. Las conclusiones han cambiado a lo largo del tiempo, con los cambios de la tecnología y el estilo de diseño de las máquinas. El soporte hardware tiene la ventaja de la velocidad, pero mover funcionalidad al software tiene la ventaja de la flexibilidad y la adaptabilidad a diferentes situaciones. Los trabajos de Dijkstra y Knuth mostraron que es posible proporcionar exclusión mutua únicamente con operaciones de lectura y escritura exclusiva (suponiendo una memoria con consistencia secuencial). Sin embargo, todas las operaciones de sincronización prácticas descansan en primitivas atómicas del tipo *leer-modificar-escribir*, en donde el valor de una posición de la memoria se lee, modifica y se vuelve a escribir de manera atómica.

La historia del diseño del conjunto de instrucciones nos permite observar la evolución de soporte hardware para la sincronización. Una de las instrucciones clave fue la inclusión en el IBM 370 de una instrucción atómica sofisticada, la instrucción *compare-and-swap*, para soportar la sincronización en multiprogramación sobre sistemas uniprocesador o multiprocesador. Esta instrucción compara el valor de una posición de la memoria con el valor de un registro específico, y si son iguales se intercambia el valor de la posición con el valor de un segundo registro. El Intel x86 permite añadir a cualquier instrucción un prefijo que la hace atómica, con lo que haciendo que los operandos fuente y destino sean posiciones de la memoria es posible utilizar la mayor parte del conjunto de instrucciones para implementar operaciones atómicas. También se ha propuesto que las operaciones de sincronización a nivel de usuario, como barreras y candados, de-

ben ser soportadas a nivel de máquina, y no sólo las primitivas *leer-modificar-escribir*; es decir, el propio algoritmo de sincronización debe implementarse en hardware. Esta cuestión pasó a ser muy activa a raíz del debato sobre la reducción del conjunto de instrucciones, ya que las operaciones que acceden a memoria fueron reducidas a simple instrucciones de lectura y escritura con un único operando. La solución de SPARC fue proporcionar operaciones atómicas de intercambio entre un registro y una dirección de memoria (intercambio atómico entre el contenido de la posición de memoria y el registro) y de comparación e intercambio, mientras que MIPS renunció a las primitivas atómicas utilizadas en sus anteriores conjuntos de instrucciones, al igual que hizo la arquitectura IBM Power usada en el RS6000. La primitiva que se introdujo fue la combinación de un par de instrucciones que incluye una instrucción de lectura especial denominada *load linked* o *load locked* y una instrucción de escritura especial denominada *store conditional*. Estas instrucciones se usan como una secuencia: Si el contenido de la dirección de memoria especificada por la carga se modifica antes de que ocurra el almacenamiento condicionado, entonces este último no se realiza. Si el procesador realiza un cambio de contexto entre las dos instrucciones, tampoco se realiza el almacenamiento condicionado. El almacenamiento condicionado se define de tal manera que devuelva un valor para indicar si dicho la operación se puedo realizar con éxito. Esta aproximación ha sido posteriormente incorporada por las arquitectura PowerPC y DEC Alpha, siendo en la actualidad muy popular. En la figura 5.28 se muestra un ejemplo de cómo se pueden implementar algunas operaciones atómicas con la utilización de estas dos operaciones.

```
try:  mov   R3, R4      ; mov exchange value
      ll    R2, 0(R1)   ; load linked
      sc    R3, 0(R1)   ; store conditional
      beqz  R3, try     ; branch store fails
      mov   R4, R2      ; put load value in R4
```

```
try:  ll    R2, 0(R1)   ; load linked
      addi  R2, R2, #1  ; increment
      sc    R2, 0(R1)   ; store conditional
      beqz  R2, try     ; branch store fails
```

Figura 5.28: Implementación de (a) un intercambio atómico y (b) una operación de lectura e incremento (*fetch-and-increment*) utilizando las operaciones *load linked* y *store conditional*

Componentes de un evento de sincronización

Consideremos los componentes de un evento de sincronización, lo que nos permitirá ver claramente por qué el soporte directo en hardware de la exclusión mutua y otras operaciones similares es difícil y da lugar a implementaciones demasiado rígidas. Existen tres componentes principales para un tipo dado de evento de sincronización:

un *método de adquisición*: un método mediante el cual un proceso intenta adquirir el derecho de la sincronización (para entrar a la sección crítica o para proceder a pasar dicho evento)

un *algoritmo de espera*: un método mediante el cual un proceso espera a que una sincronización esté disponible cuando ésta no lo está. Por ejemplo, si un proceso intenta adquirir un cerrojo pero éste no está libre (o quiere proceder a pasar un evento que todavía no ha ocurrido), éste debe esperar de alguna manera hasta que el cerrojo esté libre.

un *método de liberación*: un método que permita a un proceso habilitar el paso de un evento de sincronización a otros procesos; por ejemplo, una implementación de la operación UNLOCK, un método para que el último proceso que llega a una barrera libere a los procesos que están esperando, o un método para notificar a un proceso que está esperando en un evento punto a punto que el evento ha ocurrido.

La elección del algoritmo de espera es bastante independiente del tipo de sincronización. ¿Qué debe hacer un proceso que llega al punto de adquisición mientras que espera a que ocurra una liberación? Existen dos alternativas: *espera activa* y *bloqueo*. La espera activa significa que el proceso se queda ejecutando un bucle que testea de forma repetitiva si una variable a cambiado de valor. Un liberación del evento de sincronización por parte de otro proceso cambia el valor de la variable, permitiendo proceder al proceso que estaba esperando. En el caso del bloqueo, el proceso simplemente se bloquea (suspende) indicando al procesador el evento que está esperando que ocurra. El proceso será despertado y puesto en un estado de listo para su ejecución cuando la liberación que estaba esperando ocurra. Las diferencias entre la espera activa y el bloqueo son claras. La espera activa se comporta mejor cuando el periodo de espera es pequeño, mientras que el bloqueo es preferible en el caso de que el periodo de espera sea grande y si hay otros procesos en ejecución. La dificultad de implementar operaciones de sincronización de alto nivel en hardware no reside en los componentes de adquisición y liberación, sino en el algoritmo de espera. Por esta razón, es conveniente proporcionar soporte hardware para los aspectos críticos de la adquisición y liberación y permitir que los tres componentes se unan en software.

El papel del usuario, del sistema software y del hardware

¿Quién debe ser el responsable de implementar las operaciones de sincronización de alto nivel tales como los candados y las barreras?. Normalmente, un programador quiere usar candados, eventos, o incluso operaciones de más alto nivel sin tenerse que preocupar de su implementación interna. La implementación se deja en manos del sistema, que debe decidir cuánto soporte hardware debe proporcionar y cuánta funcionalidad implementar en software. Se han desarrollado algoritmos de sincronización software que utilizan simples primitivas atómicas de intercambio y que se aproximan a la velocidad proporcionada por las implementaciones basadas totalmente en hardware, lo que los hace muy atractivos. Como en otros aspectos de diseño de un sistema, la utilización de operaciones más rápidas depende de la frecuencia de utilización de estas operaciones en las aplicaciones. Así, una vez más, la respuesta estará determinada por un mejor entendimiento del comportamiento de la aplicación.

Las implementaciones software de las operaciones de sincronización suelen incluirse en librerías del sistema. Muchos sistemas comerciales proporcionan subrutinas o llamadas al sistema que implementan cerrojos, barreras e incluso algún otro tipo de evento de sincronización. El diseño de una buena librería de sincronización puede suponer un desafío. Una complicación potencial es que el mismo tipo de sincronización, e incluso la misma variable de sincronización, puede ejecutarse en diferentes ocasiones bajo condiciones de ejecución muy diferentes. Estos escenarios diferentes imponen requisitos en el rendimiento muy diferentes. Bajo una situación de alta contención, la mayoría de los procesos gastarán su tiempo esperando por lo que el requisito clave será que el algoritmo proporcione un gran ancho de banda para las operaciones de lock-unlock,

mientras que en condiciones de baja carga la meta es proporcionar baja latencia para la adquisición del cerrojo. Una segunda complicación es que los multiprocesadores se usan normalmente con cargas de multiprogramación donde la planificación de los procesos y otras interacciones entre los recursos pueden cambiar el comportamiento de los procesos de un programa paralelo en cuanto a la sincronización. Todos estos aspectos hacen de la sincronización un punto crítico en la interacción hardware/software.

5.5.1. Cerrojos (exclusión mutua)

La operaciones de exclusión mutua (lock/unlock) se implementan utilizando un amplio rango de algoritmos. Los algoritmos simples tienden a ser más rápidos cuando existen poca contención, pero son ineficientes ante altas contenciones, mientras que los algoritmos sofisticados que se comportan bien en caso de contención tienen un mayor coste en el caso de baja contención.

Implementación de cerrojos usando coherencia

Antes de mostrar los distintos mecanismos de implementación de la exclusión mutua es conveniente articular algunas metas de rendimiento que perseguimos en el caso de los cerrojos. Estas metas incluyen:

Baja latencia. Si un cerrojo está libre y ningún otro procesador está tratando de adquirirlo al mismo tiempo, un procesador debe ser capaz de adquirirlo con baja latencia.

Bajo tráfico. Supongamos que varios o todos los procesadores intentan adquirir un cerrojo al mismo tiempo. Debería ser posible la adquisición del cerrojo de forma consecutiva con tan poca generación de tráfico o transacciones de bus como sea posible. Un alto tráfico puede ralentizar las adquisiciones debido a la contención, y también puede ralentizar transacciones no relacionadas que compiten por el bus.

Escalabilidad. En relación con el punto anterior, ni la latencia ni el tráfico deben variar más rápidamente que el número de procesadores usados. Hay que tener en cuenta que dado que el número de procesadores en un SMP basado en bus no es probable que sea grande, no es importante la escalabilidad asintótica.

Bajo coste de almacenamiento. La información que es necesaria debe ser pequeña y no debe escalar más rápidamente que el número de procesadores.

Imparcialidad. Idealmente, los procesadores deben adquirir un cerrojo en el mismo orden que sus peticiones fueron emitidas. Al menos se debe evitar la muerte por inanición o una imparcialidad excesiva.

La primera implementación que veremos es la implementación de los *spin locks*: cerrojos que el procesador intenta adquirir continuamente ejecuanto un bucle. Como ya se comentó, este tipo de cerrojos se usan cuando se espera que el tiempo que se va a estar esperando es pequeño.

Una primera implementación de un cerrojo de este tipo sería el que se presenta en el siguiente código:

```
        li     R2,#1
lockit: exch   R2,0(R1)
        bnez   R2,lockit
```

Una de las ventajas de esta implementación se produce en el caso de que exista una localidad en el acceso al cerrojo: es decir, el procesador que ha usado el cerrojo en último lugar es probable que lo use de nuevo en un futuro cercano. En estos casos, el valor del cerrojo ya reside en la caché del procesador, reduciendo el tiempo necesario para adquirir el cerrojo.

Otra ventaja que nos permite la coherencia es que el test y la adquisición del cerrojo se pueda realizar sobre la copia local sin necesidad de transacciones de bus, reduciendo el tráfico del bus y los problemas de congestión. Para conseguir esta ventaja es necesario realizar un cambio en la implementación. Con la implementación actual, cada intento de intercambio requiere una operación de escritura. Si varios procesadores intentan obtener el cerrojo, cada uno generará una escritura. La mayoría de estas escrituras darán lugar a fallos de escritura, dado que cada procesador está intentando obtener la variable cerrojo en un estado exclusivo.

Así, podemos modificar la implementación de tal manera que el bucle dé lugar a lecturas sobre una copia local del cerrojo hasta que observe que está disponible. El procesador primero lee el valor de la variable para testear su estado. Un procesador sigue leyendo y testeando hasta que el valor de la lectura indica que el cerrojo está libre. El procesador después compite con el resto de procesadores que también pudiesen estar esperando. Todos los procesos utilizan una instrucción de intercambio que lee el valor antiguo y almacena un 1 en la variable cerrojo. El ganador verá un 0, y los perdedores un 1 en la variable cerrojo. El procesador ganador ejecuta el código existente después de la adquisición del cerrojo y, cuando termina, almacena un 0 en la variable cerrojo para liberarlo, con lo que comienza de nuevo la carrera para adquirirlo. A continuación se muestra el código del cerrojo mejorado:

```
lockit: lw     R2,0(R1)
        bnez   R2,lockit
        li     R2,#1
        exch   R2,0(R1)
        bnez   R2,lockit
```

Examinemos cómo este esquema usa el mecanismo de coherencia de la caché. La figura 5.29 muestra el procesador y las operaciones de bus cuando varios procesos intentan adquirir la variable cerrojo usando un intercambio atómico. Una vez que el procesador que tiene el cerrojo lo libera (almacena un 0), las demás cachés son invalidadas y deben obtener el nuevo valor para actualizar sus copias de la variable. Una de esas cachés obtiene la copia con el nuevo valor del cerrojo (0) y realiza el intercambio. Cuando se satisface el fallo de caché para los otros procesadores, encuentran que la variable ya está adquirida, así que vuelven al bucle de testeo.

Este ejemplo muestra otra ventaja de las primitivas *load-linked/store-conditional*: las operaciones de lectura y escritura están explícitamente separadas. La lectura no necesita producir ningún tráfico en el bus. Esto permite la siguiente secuencia de código que tiene las mismas características que la versión de intercambio optimizada (R1 contiene la dirección del cerrojo):

```
lockit: ll     R2,0(R1)
```

Step	Processor P0	Processor P1	Processor P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back from P0
4		(Waits while bus/ directory busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets Lock =1	Exclusive	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1	Enter critical section	Shared	Bus/directory services P2 cache miss; generates write back
8		Spins, testing if lock = 0			None

Figura 5.29: Pasos en la coherencia de la caché y tráfico en el bus para tres procesadores, P0, P1 y P2 que intentan adquirir un cerrojo en protocolo de coherencia por invalidación.

```
bnez   R2,lockit
li     R2,#1
sc     R2,0(R1)
beqz   R2,lockit
```

El primer bucle comprueba el cerrojo, mientras que el segundo resuelve las carreras cuando dos procesadores ven el cerrojo disponible de forma simultánea.

Aunque nuestro esquema de cerrojo es simple y convincente, tiene el problema a la hora de escalar debido al tráfico generado cuando se libera el cerrojo. Para entender por qué no escala bien, imaginemos una máquina con todos los procesadores compitiendo por el mismo cerrojo. El bus actúa como un punto de serialización para todos los procesadores, dando lugar a mucha contención. El siguiente ejemplo muestra cómo pueden ir de mal las cosas.

Ejemplo Supongamos 20 procesadores en un bus, cada uno de ellos intentando obtener el acceso en exclusiva sobre una variable. Supongamos que cada transacción del bus (fallo de lectura o de escritura) tarda 50 ciclos. Vamos a ignorar el tiempo de acceso a la caché, así como el tiempo que el cerrojo permanece cerrado. Para determinar el número de transacciones de bus necesarias para que los 20 procesadores adquieran el cerrojo, suponiendo que todos están en el bucle principal cuando se libera el cerrojo en el tiempo 0. ¿Cuánto se tardará en procesar las 20 peticiones? Supongamos que el bus es totalmente justo de tal manera que cualquier petición pendiente se sirve antes de cualquier nueva petición y que los procesadores son iguales de rápidos.

Respuesta La figura 5.30 muestra la secuencia de eventos desde que se libera el cerrojo

hasta la siguiente vez que se libera. Por supuesto, el número de procesadores compitiendo por el cerrojo desciende cada vez que se adquiere el mismo, lo que reduce el coste medio a 1525 ciclos. Por lo tanto, para 20 pares de eventos lock-unlock será necesario más de 30.000 ciclos para que todos los procesadores adquieran el cerrojo. Además, un procesador estará esperando, en promedio, la mitad de este tiempo sin hacer nada, simplemente intentando obtener la variable. El número de transacciones de bus generadas supera las 400.

Event	Duration
Read miss by all waiting processors to fetch lock (20×50)	1000
Write miss by releasing processor and invalidates	50
Read miss by all waiting processors (20×50)	1000
Write miss by all waiting processors, one successful lock (50), and invalidation of all lock copies (19×50)	1000
Total time for one processor to acquire and release lock	3050 clocks

Figura 5.30: Tiempo para adquirir y liberar un cerrojo cuando 20 procesadores están compitiendo por el mismo.

La raíz del problema está en la contención y en el hecho de que los accesos se serialicen. Para solucionar estos problemas vamos a estudiar una serie de algoritmos avanzados que intentarán tener baja latencia en el caso de poca o nula contención y que minimice la serialización en el caso de que la contención sea significativa.

Algoritmos avanzados para cerrojos

Como ya se ha comentado es deseable tener un único proceso intentando obtener un cerrojo cuando éste se libera (más que permitir que todos los procesadores intenten adquirirlo a la vez). Sería incluso más deseable que únicamente un procesador incurriera en un fallo de lectura cuando se produce una liberación. En primer lugar veremos una modificación del cerrojo optimizado que reduce la contención retrasando artificialmente los procesos cuando fallan en la adquisición de un cerrojo. Posteriormente veremos dos algoritmos: *ticket lock* (que garantiza la primera afirmación) y *array-based lock* (que garantiza ambas) y que aseguran que el acceso al cerrojo se realizará en una ordenación FIFO.

Spin lock con *exponential back-offs*. Este método opera retrasando artificialmente los procesos cuando no obtienen el cerrojo. El mejor rendimiento se obtiene cuando este retraso crece de forma exponencial en función del número de intentos para adquirir el cerrojo. La figura 5.31 muestra una implementación de este algoritmo. Esta implementación intenta preservar una baja latencia cuando la contención es pequeña no retrasando la primera iteración.

Ticket Lock. Este algoritmo actúa como un sistema de ticket. Cada proceso que espera la adquisición de un cerrojo toma un número, y realiza una espera activa sobre un número global que indica a quién se está sirviendo en la actualidad —como el número que muestra la pantalla luminosa en un número de espera— hasta que el número global

```
        li    R3,1        ;R3 = initial delay
lockit: ll    R2,0(R1)    ;load linked
        bnez  R2,lockit   ;not available-spin
        addi  R2,R2,1     ;get locked value
        sc    R2,0(R1)    ;store conditional
        bnez  R2,gotit    ;branch if store succeeds
        sll   R3,R3,1     ;increase delay by 2
        pause R3          ;delays by value in R3
        j     lockit
goit:   use data protected by lock
```

Figura 5.31: Un cerrojo con *exponential back-off*.

es igual al número que ha obtenido. Para liberar el cerrojo, un proceso simplemente incrementa el número global. La primitiva atómica necesaria es del tipo *fetch&increment*, que utiliza un proceso para obtener su ticket del contador compartido. Esta primitiva puede implementarse mediante una instrucción atómica o usando LL-SC. Este cerrojo presenta una sobrecarga en el caso de ausencia de contención similar a la del cerrojo LL-SC. Sin embargo, al igual que el cerrojo LL-SC presenta todavía un problema de tráfico. La razón es que todos los procesadores iteran sobre la misma variable. Cuando la variable se modifica en la liberación, todas las copias en las cachés se invalidan e incurren en un fallo de lectura. (El cerrojo LL-SC presentaba un comportamiento un poco peor, ya que en ese caso ocurría otra invalidación y un conjunto de lecturas cuando un procesador tenía éxito en su SC). Una manera de reducir este tráfico es introducir algún tipo de espera. No queremos usar una espera exponencial ya que no queremos que todos los procesos esperen cuando se libere el cerrojo y así ninguno intente adquirirlo durante algún tiempo. Un técnica de compromiso es que cada procesador retrase la lectura del contador una cantidad proporcional al tiempo que estiman que tardará en llegarle su turno; es decir, una cantidad proporcional a la diferencia entre su ticket y el número del contador en su última lectura.

Array-based Lock. La idea aquí es utilizar una instrucción *fetch&increment* para obtener no un valor sino una localización única sobre la que realizar la espera activa. Si existen p procesadores que pueden completar un cerrojo, entonces el cerrojo contiene un array con p localizaciones sobre las cuales los procesos pueden iterar, idealmente cada una en un bloque de memoria separado para evitar la falsa compartición. El método de adquisición es usar una operación *fetch&increment* para obtener la siguiente dirección disponible en el array sobre la cual iterar, el método de espera es iterar sobre esta localización, y el método de liberación consiste en escribir un valor que indica "liberado" sobre la siguiente localización en el array. Sólo el procesador que estaba iterando sobre esa localización tiene en su caché un bloque invalidado, y un posterior fallo de lectura le indica que ha obtenido el cerrojo. Como en el *ticket lock*, no es necesario realizar una operación de testeo después del fallo ya que únicamente se notifica a un proceso la liberación del cerrojo. Este cerrojo es claramente FIFO y por lo tanto justo. Su latencia en el caso de ausencia de contención es similar a de los cerrojos vistos hasta ahora y es más escalable que el algoritmo anterior ya que solamente un proceso incurre en un fallo de lectura. El único inconveniente es que el espacio usado es $O(p)$ en lugar de $O(1)$, pero dado que p y la constante de proporcionalidad son pequeños el problema no es

significativo. También presenta un inconveniente potencial en el caso de máquinas de memoria distribuida.

Algoritmos para cerrojos en sistemas escalables

Al principio de esta sección se han discutido los distintos tipos de cerrojos existentes, cada uno de los cuales daba un paso más en la reducción del tráfico del bus y de la equidad, pero a menudo a costa de una mayor sobrecarga. Por ejemplo, el *ticket lock* permite que únicamente un procesador pida el cerrojo cuando éste se libera, pero la notificación de la liberación llega a todos los procesadores a través de una invalidación y el posterior fallo de lectura. El cerrojo basado en array soluciona este problema haciendo que cada proceso espere sobre diferentes localizaciones, y el procesador que libera el cerrojo únicamente notifica la liberación a un procesador escribiendo en la localización correspondiente.

Sin embargo, las cerrojos basados en array tienen dos problemas potenciales en máquinas escalables con memoria física distribuida. En primer lugar, cada cerrojo requiere un espacio proporcional al número de procesadores. En segundo lugar, y más importante para las máquinas que no permiten tener en caché datos remotos, no hay forma de saber por anticipado sobre qué localización iterará un proceso concreto, dado que dicha localización se determina en tiempo de ejecución a través de una operación *fetch&increment*. Esto hace imposible almacenar las variables de sincronización de manera que la variable sobre la cual itera un proceso esté siempre en su memoria local (de hecho, todos los cerrojos del capítulo anterior tienen este problema). En máquinas de memoria distribuida sin coherencia de cachés, como la Cray T3D y T3E, éste es un gran problema, ya que los procesos iterarían sobre localizaciones remotas, dando lugar a una gran cantidad de tráfico y contención. Afortunadamente, existe un algoritmo software que reduce la necesidad de espacio al tiempo que asegura que todas las iteraciones se realizarán sobre variables almacenadas localmente.

Software Queuing Lock. Este cerrojo es una implementación software de un cerrojo totalmente hardware propuesto inicialmente en el proyecto Wisconsin Multicube. La idea es tener una lista distribuida o cola de procesos en espera de la liberación del cerrojo. El nodo cabecera de la lista representa el proceso que tiene el cerrojo. Los restantes nodos son procesos que están a la espera, y están la memoria local del proceso. Un nodo apunta al proceso (nodo) que ha intentado adquirir el cerrojo justo después de él. También existe un puntero cola que apunta al último nodo de la cola, es decir, el último nodo que ha intentado adquirir el cerrojo. Veamos de una manera gráfica como cambia la cola cuando los procesos adquieren y liberan el cerrojo.

Supongamos que el cerrojo de la figura 5.32 está inicialmente libre. Cuando un proceso A intenta adquirir el cerrojo, lo obtiene y la cola quedaría como se muestra en la figura 5.32(a). En el paso (b), el proceso B intenta adquirir el cerrojo, por lo que se pone a la cola y el puntero cola ahora apunta a él. El proceso C es tratado de forma similar cuando intenta adquirir el cerrojo en el paso (c). B y C están ahora iterando sobre las variables locales asociadas con sus nodos en la cola mientras que A mantiene el cerrojo. En el paso (d), el proceso A libera el cerrojo. Después "despierta" al siguiente proceso, B, de la cola escribiendo en la variable asociada al nodo B, y dejando la cola. B tiene ahora el cerrojo, y es la cabeza de la cola. El puntero cola no cambia. En el paso (e), B libera el cerrojo de manera similar, pasándoselo a C. Si C libera el cerrojo antes de que otro proceso intente adquirirlo, el puntero al cerrojo será NULL

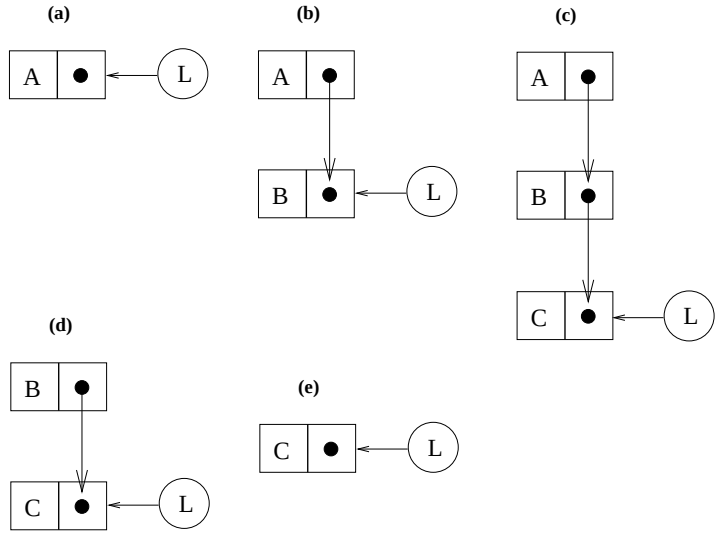


Figura 5.32: Estados de la lista para un cerrojo cuando los procesos intentan adquirirlo y cuando lo liberan.

y el cerrojo estará otra vez libre. De esta manera, los procesos tienen garantizado un acceso al cerrojo en una ordenación FIFO con respecto al orden en el que intentaron adquirirlo.

Debe quedar claro que en el cerrojo por cola software el espacio necesario es proporcional al número de procesos que están esperando o participando en el cerrojo, no al número de procesos en el programa. Este sería el cerrojo que se elegiría para máquinas que soportan un espacio de memoria compartido con memoria distribuida pero sin coherencia de cachés.

En el caso de máquinas con coherencia de caché, al igual que con el resto de algoritmos que veamos, se pueden aplicar los mismos algoritmos teniendo en cuenta dos diferencias. Por una parte, las implicaciones de rendimiento de iterar sobre variables remotas son menos significativas, dada la posibilidad de tener una copia de la variable en la caché local. El que cada procesador itere sobre una variable distinta es, por supuesto, útil ya que de esta forma no todos los procesadores deben ir al mismo nodo origen para solicitar el nuevo valor de la variable cuando esta se invalida, reduciendo la contención. Sin embargo, existe una razón (muy poco probable) que puede ser muy importante para el rendimiento en el caso de que la variable sobre la cual itera un procesador esté almacenada localmente: si la caché está unificada, es de asignación directa y las instrucciones de bucle de iteración entran en conflicto con la propia variable, en cuyo caso los fallos debido a conflicto se satisfacen localmente. Un beneficio menor de una buena ubicación es convertir los fallos que ocurren después de una invalidación en fallos de dos saltos en lugar de tres.

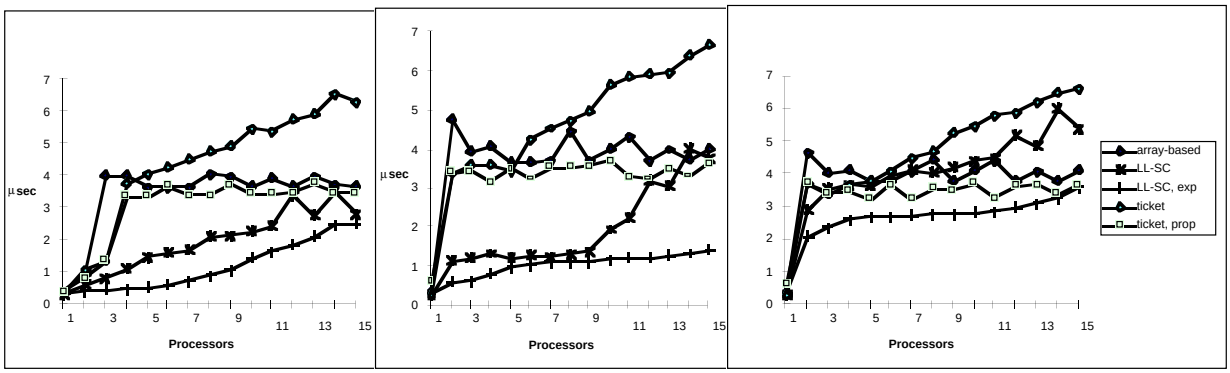
Rendimiento de los cerrojos

A continuación examinaremos el rendimiento de diferentes cerrojos en el SGI Challenge, como se muestra en las figura 5.33. Todos los cerrojos se han implementado usando LL-SC, ya que Challenge únicamente proporciona estas instrucciones. El cerrojos *test&set* se han implementado simulando la instrucción test&set mediante instruc-

ciones LL-SC. En particular, cada vez que una SC falla en una escritura realizada sobre otra variable en el mismo bloque de caché, causa una invalidación al igual que haría la instrucción test&set. Los resultados se muestran en función de dos parámetros: (c) duración de la sección crítica y (d) tiempo entre la liberación del cerrojo y la siguiente vez que se intenta adquirirlo. Es decir, el código es un bucle sobre el siguiente cuerpo:

lock(L); critical_section(c); unlock(L); delay(d)

Vamos a considerar tres casos —(i) $c = 0$, $d = 0$, (ii) $c = 3{,}64\mu s$, $d = 0$, y (iii) $c = 3{,}64\mu s$, $d = 1{,}29\mu s$— denominados *nulo, sección crítica* y *retraso*, respectivamente. Recordar que en todos los casos c y d (multiplicado por el número de adquisiciones de cerrojo por cada procesador) se substrae del tiempo total de ejecución.



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figura 5.33: Rendimiento de los cerrojos en el SGI Challenge para tres posibles escenarios.

Consideremos el caso de sección crítica nula. En este caso se observa la implementación LL-SC básica del cerrojo se comporta mejor que las más sofisticadas *ticket lock* y *array-based lock*. La razón del mejor comportamiento de los cerrojos LL-SC, particularmente para pocos procesadores, es que no son equitativos, y esta falta de equidad se explota en las interacciones a nivel de arquitectura. En particular, cuando un procesador que realiza una escritura para liberar un cerrojo realiza inmediatamente una lectura (LL) para su siguiente adquisición, es muy probable que la lectura y la SC tengan éxito en su caché antes de que otro procesador pueda leer el bloque en el bus.(En el caso del Challenge la diferencia es mucho más grande ya que el procesador que ha liberado el cerrojo puede satisfacer su siguiente lectura a través de su buffer de escritura antes incluso de que la correspondiente lectura exclusiva llegue al bus). La transferencia del cerrojo es muy rápida, y el rendimiento es bueno. Al incrementarse el número de procesadores, la probabilidad de autotransferencia desciende y el tráfico en el bus debido a las invalidaciones y a fallos de lectura se incrementa, por lo que también se incrementa el tiempo entre trasferencias del cerrojo. El retraso exponencial ayuda a reducir el tráfico en ráfagas y por tanto disminuye el ratio de escalada.

Al otro extremo ($c = 3{,}64\mu s$, $d = 1{,}29\mu s$), observamos que el cerrojo LL-SC no tiene un buen comportamiento, incluso para pocos procesadores. Esto es debido a que el procesador realiza una espera después de liberar el cerrojo y antes de intentar adquirirlo de nuevo, haciendo mucho más probable que otros procesadores en espera adquieran el

cerrojo antes. Las autotransferencias son escasas, de tal manera que la transferencia del cerrojo es más lenta incluso en el caso de dos procesadores. Es interesante observar que el rendimiento para el caso de retraso exponencial es particularmente malo cuando el retraso d entre la liberación y la siguiente adquisición es distinto de cero y el número de procesadores es pequeño. Esto se debe a que para pocos procesadores, es muy probable que mientras que un procesador que acaba de liberar el candado está esperando a que expire d antes de intentar su siguiente adquisición, los otros procesadores están en un periodo de espera y ni siquiera intentan obtener el cerrojo.

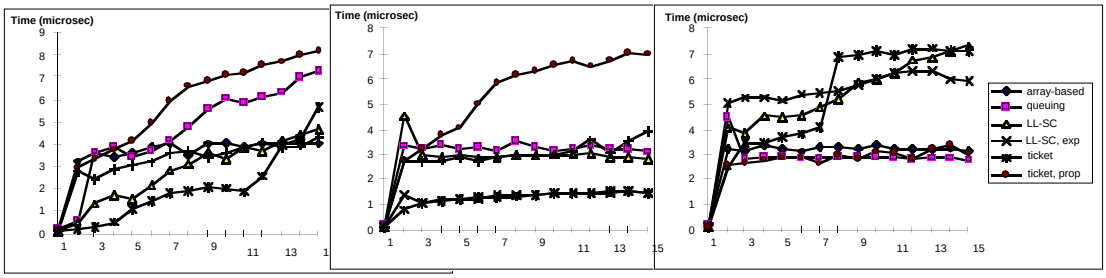
Consideremos el resto de implementaciones. Estos son equitativos, de tal manera que cada transferencia de adquisición es a un procesador diferente e implica transacciones de bus en el camino crítico de la transferencia. De aquí que todas impliquen cerca de tres transacciones de bus en el camino crítico por transferencia de cerrojo incluso cuando se usan dos procesadores. Las diferencias reales en el tiempo se deben a las transacciones de bus exactas y a la latencia que pueda ser ocultada por parte del procesador. El *ticket lock* sin periodos de espera escala pobremente: Con todos los procesos intentando leer el contador de servicio, el número esperado de transacciones entre la liberación y la lectura por parte del procesador correcto es $p/2$, dando lugar a una degradación lineal en la transferencia del cerrojo en el camino crítico. Con una espera proporcional, es probable que el procesador correcto sea el que realice la lectura en primer lugar después de la liberación, de tal manera que el tiempo por transferencia no aumenta con p. El *array-based lock* también presenta una propiedad similar, dado que únicamente el procesador correcto realiza la lectura, de tal manera que su rendimiento no se degrada con el incremento de procesadores.

Los resultados ilustran la importancia de las interacciones de la arquitectura en la determinación del rendimiento de los cerrojos, y que el cerrojo LL-SC se comporta bastante bien en buses que tienen suficiente ancho de banda (y número de procesadores por buses reales). El rendimiento para los cerrojos LL-SC no equitativos aumenta hasta ser tan mala o un poco peor que el más sofisticado de los cerrojos para más de 16 procesadores, debido al mayor tráfico, pero no demasiado debido al gran ancho de banda. Cuando se utiliza la espera exponencial para reducir el tráfico, el cerrojo LL-SC muestra el mejor tiempo de transferencia para todos los casos. Los resultados ilustran la dificultad y la importancia de la metodología experimental en la evolución de los algoritmos de sincronización. Las secciones críticas nulas muestran algunos efectos interesantes, pero una comparación significativa depende de los patrones de sincronización en la aplicaciones reales. Un experimento que utiliza LL-SC pero garantiza una adquisición en *round-robin* entre los procesos (equidad) usando una variable adicional muestra un rendimiento muy similar al *ticket lock*, confirmando que la falta de equidad y las autotransferencias son la razón del mejor rendimiento para pocos procesadores.

Rendimiento de los cerrojos para sistemas escalables

Los experimentos realizados para ilustrar el rendimiento de la sincronización son los mismos que se usaron en la sección 5.5 para el SGI Challenge, usando de nuevo LL-SC como primitiva para construir operaciones atómicas. El resultado para los algoritmos que hemos discutido se muestran en la figura 5.34 para ejecuciones con 16 procesadores sobre la máquina SGI Origin2000. De nuevo se usan tres conjuntos diferentes de valores para los retrasos dentro y después de la sección crítica.

Al igual que para el caso de memoria compartida, hasta que no se introducen retra-



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figura 5.34: Rendimiento de los cerrojos son la máquina SGI Origin2000, para tres escenarios diferentes.

sos entre secciones críticas el comportamiento de los cerrojos no equitativos presenta un mayor rendimiento. El retraso exponencial ayuda al LL-SC cuando existe una sección crítica nula, dado que en este caso es posible reducir la alta contención que se produce. En caso contrario, con secciones criticas nulas el cerrojo basado en ticket escala pobremente, aunque en el caso de usar retraso exponencial escala muy bien. Los cerrojos basados en arrays también escalan muy bien. En el caso de existir coherencia de cachés, el mejor emplazamiento de las variables cerrojo en la memoria principal proporcionada por el cerrojo por colas software no es particularmente útil, y de hecho incurre en contención al realizar las operaciones de comparación e intercambio (implementadas con LL-SC) y escala peor que el cerrojo basado en array. Si forzamos a los cerrojos a que tengan un comportamiento equitativo, su comportamiento es muy parecido al cerrojo basado en ticket sin retraso exponencial.

Si usamos una sección crítica no nula y un retraso entre los accesos al cerrojo (Figura 5.34(c)), suponiendo que todos los cerrojos se comportan equitativamente, el cerrojo LL-SC pierde su ventaja, mostrando su peor escalamiento. El cerrojo basado en array, el cerrojo por colas, y el cerrojo basado en ticket con retraso exponencial escalan muy bien. La mejor colocación de los datos del cerrojo por colas no importa, pero su contención no es peor que la del resto. En resumen, podemos concluir que el cerrojo basado en array y en ticket presentan un buen comportamiento y robustez para máquinas escalables con coherencia de caché, al menos al ser implementadas con LL-SC, y el cerrojo LL-SC con retraso exponencial se comporta mejor cuando no existe retraso entre la liberación y la siguiente adquisición, debido a su repetido comportamiento no equitativo. El cerrojo más sofisticado, el basado en cola, es innecesario pero también se comporta bien cuando existen retrasos entre la liberación y la siguiente adquisición.

5.5.2. Eventos de sincronización punto a punto (banderas)

La sincronización punto a punto dentro de un programa paralelo se implementa a menudo usando una espera activa sobre variables ordinarias que actúan como banderas (*flags*). Si queremos utilizar bloqueo en lugar de espera activa, podemos utilizar semáforos de una manera similar a la utilizada en la programación concurrente y en los sistemas operativos.

Las banderas son variables de control, usadas típicamente para comunicar que ha ocurrido un evento de sincronización más que para transferir valores. Si dos procesos tienen una relación productor/consumidor en la variable compartida a, entonces se puede utilizar una bandera para manejar la sincronización como se muestra a continuación:

P1	P2
a=f(x); /*set a */	while (flag is 0) do nothing;
flag=1;	b=g(a); /* use a */

Si sabemos que la variable *a* se ha inicializado a un cierto valor, digamos 0, la cambiaremos a un nuevo valor si estamos interesados en este evento de producción, después podemos usar el propio *a* como variable de sincronización, como sigue:

P1	P2
a=f(x); /*set a */	while (a is 0) do nothing;
	b=g(a); /* use a */

Esto elimina la necesidad de una variable de bandera (*flag*) separado, y evita la escritura y la lectura de la variable.

5.5.3. Eventos globales de sincronización (barreras)

Una operación de sincronización común en los programas con bucles paralelos es la *barrera*. Una barrera fuerza a todos los procesos a esperar hasta que todos ellos alcanzan la barrera y entonces se permite la continuación de los mismos.

Algoritmos software

Los algoritmos software para las barreras se implementan usando cerrojos, contadores compartidos y banderas. Una implementación típica de la misma se puede conseguir con un cerrojo que protege el contador que lleva la cuenta de los procesos que llegan a a la barrera. La figura 5.35 muestra una implementación típica de este tipo de barrera, denominado *barrera centralizada*.

```
LOCK(counterlock);        /* Garantiza actualización atómica */
if (count==0) release=0;  /* el primero bloquea al resto     */
count=count+1;            /* cuenta los que van llegando     */
UNLOCK(counterlock);      /* libera el cerrojo del contador  */
if (count==total)         /* Si es el último en llegar */
{
    count=0;              /* reinicia la cuenta y */
    release=1;            /* activa el release */
}
else while (release==0);  /* sino, se espera a que se active el release */
```

Figura 5.35: Código de una barrera centralizada.

En la práctica, se realiza una implementación de la barrera un poco más compleja. Frecuentemente una barrera se usa dentro de un bucle, de tal manera que los procesos salen de la barrera para realizar algún trabajo para después alcanzar la barrera de nuevo. Supongamos que uno de los procesos nunca deja la barrera (quedándose en la operación spin), lo que puede ocurrir si el SO decide en ese momento ejecutar otro proceso. Sería posible que uno de los procesos siguiera adelante y llegara a la barrera antes de que el

último proceso la hubiera dejado. El proceso rápido atrapa al proceso lento en la barrera al resetear la bandera *release*. Ahora todos los procesos esperarán indefinidamente ya que el número de procesos nunca alcanzará el valor total. La observación importante es que el programador no hizo nada mal. Es la implementación de la barrera la que hizo suposiciones que no eran correctas. Una solución obvia es contar los procesos cuando salen de la barrera (al igual que hacemos en la entrada) y no permitir la reentrada de un proceso hasta que todos los procesos hayan dejado la instancia anterior de la barrera. Esto incrementaría significativamente la latencia de la barrera y la contención, que como veremos posteriormente ya es bastante grande. Una solución alternativa es usar una *sense-reversing barrier*, que hace uso de una variable privada por proceso, *local_sense*, que se inicializa a 1 para cada proceso. La figura 5.36 muestra el código para este tipo de barrera.

```
local_sense=!local_sense; /* Cambia local_sense */
LOCK(counterlock);        /* Garantiza actualización atómica */
count=count+1;            /* cuenta los que van llegando     */
UNLOCK(counterlock);      /* libera el cerrojo del contador  */
if (count==total)                 /* Si es el último en llegar */
{
    count=0;                      /* reinicia la cuenta y */
    release=local_sense;          /* activa el release */
}
else while (release!=local_sense); /* sino, se espera a que se active el release */
```

Figura 5.36: Código para una barrera sense-reversing.

Algoritmos para barreras en sistemas escalables

El problema de eventos globales como las barreras es un preocupación clave en las máquinas escalables. Una cuestión de considerable debate es si es necesario un soporte hardware especial para las operaciones globales o si son suficientes los algoritmos software. El CM-5 representa una parte del espectro, con un red de "control" especial que proporciona barreras, reducciones, broadcasts y otras operaciones globales sobre un subárbol de la máquina. Cray T3D proporciona soporte hardware para las barreras, mientras que T3E no lo hace. Dado que es sencillo construir barreras que iteren sobre variables locales, la mayoría de las máquinas escalables no proporcionan un soporte hardware especial para las barreras sino que se construyen en librerías software usando primitivas de intercambio o LL-SC.

En la barrera centralizada usada en la máquinas basadas en bus, todos los procesadores usan el mismo cerrojo para incrementar el mismo contador cuando indican su llegada a la barrera, y esperan sobre la misma variable de flag hasta que son liberados. En una máquina más grande, la necesidad de que todos los procesadores accedan al mismo cerrojo y lean y escriban de las mismas variables puede dar lugar a mucho tráfico y contención. De nuevo, esto es particularmente cierto en el caso de máquinas sin coherencia de cachés, donde la variable se puede convertir en un punto caliente de la red cuando varios procesadores iteran sobre ella.

Es posible implementar la llegada y la salida de la barrera de una manera más distribuida, donde no todos los procesadores tienen que acceder a la misma variable o cerrojo. La coordinación de la llegada o liberación puede realizarse en fases o rondas, con subconjuntos de procesos que se coordinan entre ellos, de tal manera que después de algunas

fases todos los procesos están sincronizados. La coordinación de diferentes subconjuntos puede realizarse en paralelo, ya que no es necesario mantener ninguna serialización entre ellos. En una máquina basada en bus, distribuir las acciones de coordinación no importa mucho ya que el bus serializa todas las acciones que requieren comunicación; sin embargo, puede ser muy importante en máquinas con memoria distribuida e interconectada donde diferentes subconjuntos pueden coordinarse en diferentes partes de la red. Veamos algunos de estos algoritmos distribuidos.

Árboles de combinación software. Una manera sencilla de coordinar la llegada o partida es a través de una estructura de árbol. Un árbol de llegada es un árbol que usan los procesadores para indicar su llegada a la barrera. Se reemplaza un único cerrojo y contador por un árbol de contadores. El árbol puede ser de cualquier tipo, por ejemplo k-ario. En el caso más simple, cada hoja del árbol es un proceso que participa en la barrera. Cuando un proceso llega a la barrera, indica su llegada realizando una operación *fetch&increment* sobre el contador asociado a su padre. Después chequea que el valor devuelto por la operación para comprobar si es el último de los hermanos en llegar. Si no, simplemente espera a la liberación. En caso contrario, el proceso pasa a ser el representante de sus hermanos en el siguiente nivel del árbol y realiza una operación *fetch&increment* sobre el contador de ese nivel. De esta manera, cada nodo del árbol envía un único representante al primer nivel del árbol donde todos los procesos están representados por ese nodo hijo que ha llegado. Para un árbol de grado k, será necesario pasar por $log_k p$ niveles para completar notificación de llegada de los p procesos. Si los subárboles de procesadores están repartidos en diferentes partes de la red, y si las variables de los nodos del árbol están distribuidas de manera apropiada en la memoria, las operaciones *fetch&increment* sobre nodos que no tiene relación ascendiente-descendiente no necesitan serialización.

Se puede utilizar una estructura de árbol similar para la liberación, de tal manera que los procesadores no realicen una espera activa sobre la misma variable. Es decir, el último proceso en llegar a la barrera pone a uno la variable de liberación asociada con la raíz del árbol, sobre la cual están realizando una espera activa $k-1$ procesos. Cada uno de los k procesos pone a uno la variable de liberación del siguiente nivel del árbol sobre la que están realizando espera activa $k-1$ procesos, y así bajando en el árbol hasta que todos los procesos se liberan. La longitud del camino crítico de la barrera en términos del número de operaciones dependientes o serializadas (es decir, transacciones de red) es por tanto $O(log_k p)$, frente al $O(p)$ para una la barrera centralizada u $O(p)$ para cualquier barrera sobre un bus centralizado.

Barreras en árbol con iteración local. Existen dos formas de asegurarse de que un procesador itere sobre una variable local. Una es predeterminar qué procesador se moverá desde un nodo a su padre en el árbol basándose en el identificador del proceso y el número de procesadores que participan en la barrera. En este caso, un árbol binario facilita la implementación de la iteración local, ya que la variable sobre la que se itera puede almacenarse en la memoria local del proceso en espera en lugar de aquel que pasa al nivel del padre. De hecho, en este caso es posible implementar la barrera sin operaciones atómicas, usando únicamente operaciones de lectura y escritura como sigue. En la llegada, un proceso que llega a cada nodo simplemente itera sobre la variable de llegada asociada a ese nodo. El otro proceso asociado con el nodo simplemente escribe en la variable al llegar. El proceso cuyo papel sea iterar ahora simplemente itera sobre la variable liberada asociada a ese nodo, mientras que el otro proceso procede a ir al nodo padre. A este tipo de barrera de árbol binario se le denomina "barrera por torneo", dado que uno de los procesos puede pensarse como el perdedor del torneo en cada etapa

del árbol de llegada.

Otra forma de asegurar la iteración local es usar árboles p-nodo para implementar una barrera entre los p procesos, donde cada nodo del árbol (hoja o interno) se asigna a un único proceso. Los árboles de llegada y la activación pueden ser los mismos, o pueden mantenerse como árboles con diferentes factores de ramaje. Cada nodo interno (proceso) en el árbol mantiene un array de variables de llegada, con una entrada por hijo, almacenadas en la memoria local del nodo. Cuando un proceso llega a la barrera, si su nodo en el árbol no es una hoja entonces comprueba en primer lugar su array de flags de llegada y espera hasta que todos sus hijos hayan indicado su llegada poniendo a uno su correspondiente entrada en el array. A continuación, pone a uno su entrada en el array de llegada de su padre (remoto) y realiza una espera activa sobre el flag de liberación asociado a su nodo en el árbol de liberación. Cuando llega el proceso raíz y todas los flags de llegada de su array están activos, entonces todos los procesos han llegado. La raíz pone a uno los flags (remotos) de liberación de todos sus hijos en el árbol de liberación; estos procesos salen de su espera activa y ponen a uno los flags de sus hijos, y así hasta que todos los procesos se liberan.

Rendimiento

Las metas a alcanzar para una barrera son las misma que las que vimos para los cerrojos:

> *Baja latencia* (*longitud del camino crítico pequeño*). Ignorando la contención, nos gustaría que el número de operaciones en la cadena de operaciones dependientes necesarias para p procesadores para pasar la barrera sea pequeña.
>
> *Bajo tráfico.* Dado que las barreras son operaciones globales, es muy probable que muchos procesadores intenten ejecutar la barrera al mismo tiempo. Nos gustaría que el algoritmo reduzca el número de transacciones y por tanto una posible contención.
>
> *Escalabilidad.* En relación con el punto anterior, nos gustaría que una barrera escale bien en función del número de procesadores que vamos a usar.
>
> *Bajo coste de almacenamiento.* La información que es necesaria debe ser pequeña y no debe aumentar más rápidamente que el número de procesadores.
>
> *Imparcialidad.* Esta meta no es importante en este caso ya que todos los procesadores se liberan al mismo tiempo, pero nos gustaría asegurar que el mismo procesador no sea el último en salir de la barrera, o preservar una ordenación FIFO.

En una barrera centralizada, cada procesador accede a la barrera una vez, de tal manera que el camino crítico tiene longitud de al menos p. Consideremos el tráfico del bus. Para completar sus operaciones, una barrera centralizada de p procesadores realiza $2p$ transacciones de bus para obtener el cerrojo e incrementar el contador, dos transacciones de bus para que el último procesador ponga a cero el contador y libere la bandera, y otras $p - 1$ transacciones para leer la bandera después de que su valor haya sido invalidado. Obsérvese que éste es mejor que el tráfico para que un cerrojo sea adquirido por p procesadores, dado que en ese caso cada una de las p liberaciones dan lugar a

una invalidación dando lugar a $O(p^2)$ transacciones de bus. Sin embargo, la contención resultante puede ser sustancial si muchos procesadores llegan a la barrera de forma simultánea. En la figura 5.37 se muestra el rendimiento de algunos algoritmos de barrera.

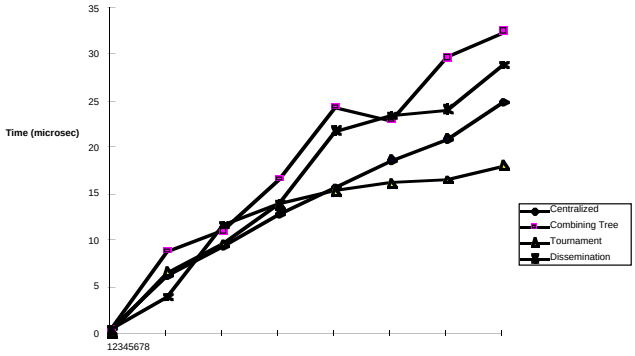


Figura 5.37: Rendimiento de algunas barreras en el SGI Challenge.

Mejora de los algoritmos de barrera en un bus

Veamos si es posible obtener una barrera con un mejor comportamiento para un bus. Una parte del problema de la barrera centralizada es que todos los procesos compiten por el mismo cerrojo y variables de bandera. Para manejar esta situación, podemos construir barreras que hagan que menos procesadores compitan por la misma variable. Por ejemplo, los procesadores pueden indicar su llegada a la barrera a través de un árbol de combinación software. En un árbol de combinación binario, únicamente dos procesadores se intercomunican su llegada a cada nodo del árbol, y únicamente uno de los dos se mueve hacia arriba para participar en el siguiente nivel del árbol. De esta forma, únicamente dos procesadores acceden a una variable dada. En una red con múltiples caminos paralelos, como los existentes en la máquinas con gran número de procesadores, un árbol de combinación se comporta mucho mejor que una barrera centralizada ya que los diferentes pares de procesadores se pueden comunicar entre ellos en paralelo por diferentes partes de la red. Sin embargo, en una interconexión centralizada como un bus, aunque cada par de procesadores se comuniquen a través de variables diferentes todos ellos generan transacciones y por tanto contención sobre el mismo bus. Dado que un árbol binario con p hojas tiene aproximadamente $2p$ nodos, un árbol de combinación necesita un número similar de transacciones de bus que la barrera centralizada. Además presenta mayor latencia ya que necesita $log\, p$ pasos para llegar de las hojas a la raíz del árbol, y de hecho del orden de p transacciones de bus serializadas. La ventaja de un árbol de combinación es que no usa cerrojos sino operaciones de lectura y escritura, que puede compensar si el número de procesadores en grande. El resto de los algoritmos de barrera escalables los estudiaremos, junto con las barreras en árbol, en el contexto de las máquinas escalables.

5.6. Generalidades sobre las redes de interconexión

La red de interconexión es uno de los elementos más importantes de cualquier arquitectura paralela puesto que va a modificar el rendimiento global del sistema y la

topología de la arquitectura. La red de interconexión va a ser el vehículo a través del cual se van a comunicar los diferentes elementos del sistema, memoria con procesador, elementos periféricos, unos procesadores con otros en un computador matricial, etc. Las redes de interconexión se utilizan en computadores matriciales así como en multiprocesadores y multicomputadores, dependiendo del tipo de sistema los elementos que se interconectan pueden variar, pero la topología de la red, así como los protocolos, suelen ser comunes e independientes del tipo de sistema. En este capítulo se estudiarán precisamente las diferentes topologías de redes de interconexión sean cuales fueren los elementos a interconectar.

En cuanto a la bibliografía de este tema, conviene consultar [DYN97] que es uno de los libros de redes para multicomputadores más completos y modernos. También en [CSG99] se encuentra una buena sección dedicada a las redes. Por último, algunas de las definiciones y parámetros se han tomado de [Wil96].

5.6.1. Definiciones básicas y parámetros que caracterizan las redes de interconexión

En esta parte dedicada a las redes se verán diferentes topologías para la construcción de redes de interconexión. La eficiencia en la comunicación en la capa de interconexión es crítica en el rendimiento de un computador paralelo. Lo que se espera es conseguir una red con latencia baja y una alta tasa de transferencia de datos y por tanto una ancho de banda amplio. Las propiedades de red que veremos a continuación van a ayudar a la hora de elegir el tipo de diseño para una arquitectura. Veamos por tanto las definiciones que nos van a caracterizar una red:

Tamaño de la red

El *tamaño de la red* es el número de nodos que contiene, es decir, el número de elementos interconectados entre si. Estos nodos pueden ser procesadores, memorias, computadores, etc.

Grado del nodo

El número de canales que entran y salen de un nodo es el *grado del nodo* y lo representaremos por *d degree*. En el caso de canales unidireccionales, el número de canales que entran en el nodo es el *grado de entrada* y los que salen el *grado de salida*. El grado del nodo representa el número de puertos de E/S y por lo tanto el coste del nodo. Esto quiere decir que el grado de los nodos debería ser lo más reducido posible para reducir costes. Si además el grado del nodo es constante se consigue que el sistema pueda ser modular y fácilmente escalable con la adición de nuevos módulos.

Diámetro de la red

El *diámetro D* de una red es el máximo de los caminos más cortos entre dos nodos cualquiera de una red.

$$D = max_{i,j \in N}(min_{p \in P_{ij}} length(p))$$

donde P_{ij} es el conjunto de caminos de i a j. La longitud del camino se mide por el número de enlaces por los que pasa el camino. El diámetro de la red nos da el número máximo de saltos entre dos nodos, de manera que de esta forma tenemos una medida de lo buena que es la comunicación en la red. Por todo esto, el diámetro de la red debería ser lo más pequeño posible desde el punto de vista de la comunicación. El diámetro se utilizó hasta finales de los 80 como la principal figura de mérito de una red, de ahí la popularidad que tuvieron en esa época redes de bajo diámetro como los hipercubos.

Anchura de la bisección

El ancho de la bisección, B, es el mínimo número de canales que, al cortar, separa la red en dos partes iguales. La bisección del cableado, B_W, es el número de cables que cruzan esta división de la red. $B_W = BW$, donde W es el ancho de un canal en bits. Este parámetro nos da una cota inferior de la densidad del cableado. Desde el punto de vista del diseñador, B_W es un factor fijo, y el ancho de la bisección restringe la anchura de los canales a $W = \frac{B_W}{B}$. La figura 5.38 muestra un ejemplo del cálculo del ancho de la bisección.

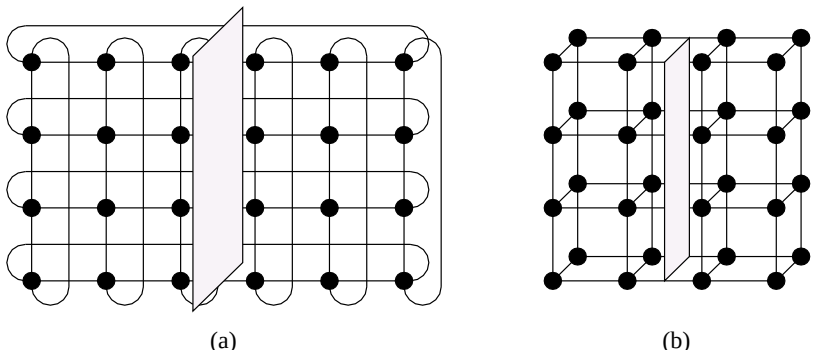


Figura 5.38: Ejemplos del cálculo del ancho de la bisección: toro 2-D. (b) toro 3-D (no se muestran los enlaces de cierre).

Cuando una red dada se divide en dos mitades iguales, al número mínimo de canales que atraviesa el corte se le llama *anchura de canal biseccional*. En el caso de una red de comunicaciones, caso común, cada canal estará compuesto por un número w de bits, hilos o cables. De esta manera podemos definir la *anchura de cable biseccional* como $B = bw$. Este parámetro B refleja la densidad de cables en una red. Esta anchura de la bisección nos da una buena indicación del ancho de banda máximo en una red a través de un corte transversal realizado en la red. El resto de cortes estarían acotados por esta anchura de la bisección.

Longitud del cable

La *longitud del cable* entre nodos es también importante puesto que tiene efectos directos sobre la latencia de la señal, desfase del reloj, o incluso los requerimientos de potencia.

Redes simétricas

Una red es *simétrica* si es isomorfa a ella misma independientemente del nodo que consideremos como origen, es decir, una red es *simétrica* si la topología de esta se ve igual desde cualquier nodo. Este tipo de redes tiene la ventaja de que simplifica mucho de los problemas de manejo de recursos. Por ejemplo, dado un patrón de tráfico uniforme, en las redes simétricas se produce una carga uniforme de los canales de la red, cosa que no ocurre en las redes asimétricas.

Otras propiedades como que si los nodos son homogéneos, o si los canales tienen memoria o son sólo conmutadores, son otras propiedades útiles para caracterizar la red.

Rendimiento de la red

Para resumir los puntos anteriormente expuestos, veamos cuales son los factores que intervienen en el rendimiento global de una red de interconexión:

Funcionalidad Esto indica cómo la red soporta el encaminamiento de datos, tratamiento de las interrupciones, sincronización, combinación petición/mensaje, y la coherencia.

Latencia de la red Indica el retraso de un mensaje, en el peor caso, a través de la red.

Ancho de banda Indica la velocidad máxima de transmisión de datos, en Mbytes/s, transmitidos a través de la red.

Complejidad hardware Indica el coste de implementación como el coste de los cables, conmutadores, conectores, arbitraje, y lógica de interfaz.

Escalabilidad Indica la capacidad de una red para expandirse de forma modular con nuevos recursos en la máquina y sin mucho detrimento en el rendimiento global.

Capacidad de transmisión de la red

La *capacidad de transmisión de una red* se define como el número total de datos que pueden ser transmitidos a través de la red por unidad de tiempo. Una forma de estimar la capacidad de transmisión es calcular la capacidad de la red, es decir, el número total de mensajes que pueden haber en la red a la vez. Normalmente, la máxima capacidad de transmisión es una fracción de su capacidad.

Un *punto caliente* suele estar formado por un par de nodos que recogen una porción demasiado grande el tráfico total de la red. El tráfico en estos puntos calientes puede degradar el rendimiento de toda la red a causa de la congestión que producen. La *capacidad de transmisión en puntos calientes* se define como la máxima velocidad a la que se pueden enviar mensajes de un nodo específico P_i a otro nodo específico P_j.

Las redes de dimensión pequeña (2D,3D) operan mejor bajo cargas no uniformes ya que se comparten más recursos. En una red de dimensiones más elevadas, los cables se asignan a una dimensión particular y no pueden ser compartidas entre dimensiones. Por ejemplo, en un n-cubo binario es posible que una línea se sature mientras otra línea físicamente adyacente, pero asignada a una dimensión diferente, permanece inactiva. En un toro, todas las líneas físicamente adyacentes se combinan en un único canal que es compartido por todos los mensajes.

La latencia mínima de la red se alcanza cuando el parámetro k y la dimensión n se eligen de manera que las componentes de la latencia debida a la distancia D (enlaces entre nodos) y a la relación L/W (longitud L del mensaje normalizada por la anchura W) quedan aproximadamente iguales. La menor latencia se obtiene con dimensiones muy bajas, 2 para hasta 10245 nodos.

Las redes de dimensión baja reducen la contención porque con pocos canales de ancho de banda amplio se consigue que se compartan mejor los recursos y, por tanto, un mejor rendimiento de las colas de espera al contrario que con muchos canales de banda estrecha. Mientras que la capacidad de la red y la latencia peor de bloque son independientes de la dimensión, las redes de dimensión pequeña tienen una mayor capacidad de transmisión máxima y una latencia de bloque media menor que las redes de dimensiones altas.

5.6.2. Topología, control de flujo y encaminamiento

El diseño de una red se realiza sobre tres capas independientes: *topología*, *encaminamiento* y *control de flujo*.

- La **topología** hace referencia al grafo de interconexión de la red $I = G(N, C)$ donde N son los nodos del grafo y C es el conjunto de enlaces unidireccionales o bidireccionales que los conectan. Si pensamos en un multiprocesador como en un problema de asignación de recursos, la topología es la primera forma de asignación de los mismos.
- El **control de flujo** hace referencia al método utilizado para regular el tráfico en la red. Es el encargado de evitar que los mensajes se entremezclen y controlar su avance para asegurar una progresión ordenada de los mismos a través de la red. Si dos mensajes quieren usar el mismo canal al mismo tiempo, el control de flujo determina (1) qué mensaje obtiene el canal y (2) qué pasa con el otro mensaje.
- El **encaminamiento** hace referencia al método que se usa para determinar el camino que sigue un mensaje desde el nodo origen al nodo destino. El encaminamiento se puede ver como una relación, $C \times N \times C$, que asigna al canal ocupado por la cabecera del mensaje y en función del nodo destino un conjunto de canales que pueden utilizarse a continuación para que el mensaje llegue a su destino. El encaminamiento es una forma dinámica de asignación de recursos. Dada una topología, un nodo actual, y un destino, la relación de encaminamiento determina como llevar un mensaje desde el nodo actual al nodo destino.

Topología

Una topología se evalúa en términos de los siguientes cinco parámetros: Ancho de la bisección, grado del nodo, diámetro de la red, longitud de la red y su simetría. Estos parámetros ya fueron expuestos al principio de este capítulo.

La topología de la red también guarda una gran relación con la construcción física de la red, especialmente con el empaquetamiento, que consiste en poner juntos todos los nodos procesadores de la red y sus interconexiones. En la figura 5.39 se muestra un ejemplo de empaquetamiento.

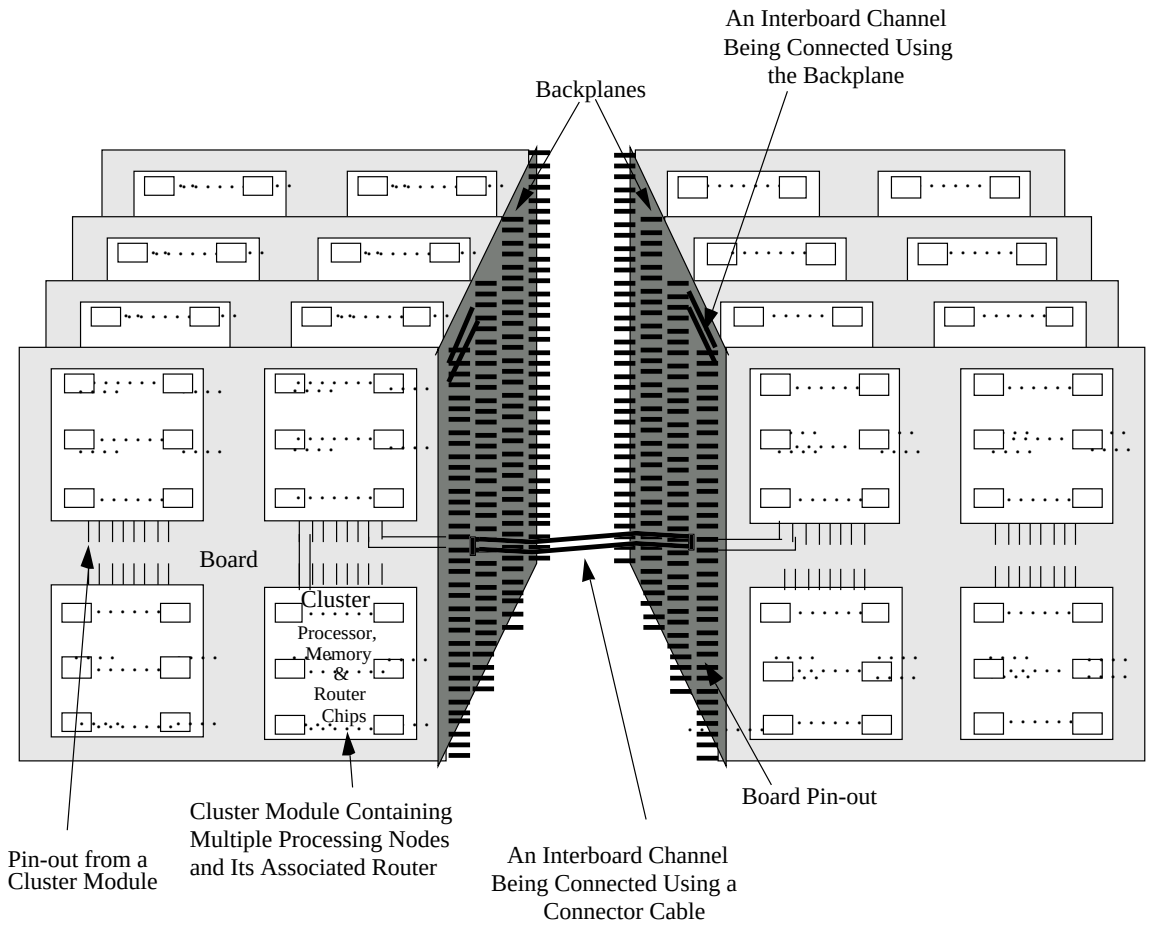


Figura 5.39: Ejemplo de empaquetamiento de un multicomputador.

Control de flujo

El control de flujo se refiere al método usado para regular el tráfico en una red. Determina cuando un mensaje o parte de un mensaje puede avanzar. También se hace cargo de la política de asignación de los recursos: buffers y canales; a las unidades de información: mensajes, paquetes y flits.

- **Mensaje**. Es la unidad lógica de comunicación. Dos objetos se comunican mediante el envío de un mensaje. Es la única unidad vista por los clientes de un servicio de red.
- **Paquete**. Un mensaje se divide en uno o más paquetes. Un paquete es la menor unidad que contiene información de encaminamiento. Si existe más de un paquete en un mensaje, cada paquete contiene además un número de secuencia que permite su reensamblaje.
- **Flit**. Un paquete se puede dividir a su vez en dígitos de control de flujo o *flits*, la menor unidad sobre la cual se puede realizar el control de flujo. Es decir, la comunicación de recursos, cables y buffers, se realizan en función de flits. En general, un flit no contiene información de encaminamiento. Únicamente el primer flit de un paquete sabe a donde se dirige. El resto de flits debe seguir al flit cabecera para determinar su encaminamiento.

Una simple analogía ilustra la diferencia entre los paquetes y los flits. Los paquetes son como los automóviles. Dado que conocen a donde van, pueden entrelazarse libremente. Los flits, por otro lado, son como los vagones de un tren. deben seguir al flit cabecera para encontrar su destino. No se pueden mezclar con los flits de otros paquetes, o perderían su único contacto con su destino.

Otro aspecto importante que tiene que solucionar el control de flujo es el de bloqueo de un paquete. En este caso será necesario disponer de algún espacio buffer para su almacenamiento temporal. Cuando ya no existe más espacio, el mecanismo de control de flujo detiene la transmisión de información. Cuando el paquete avanza y aparece más espacio buffer disponible, la transmisión comienza de nuevo. Existe también la alternativa de eliminar el paquete cuando deja de existir espacio disponible o desviarse a través de otro canal.

Encaminamiento

El encaminamiento es el método usado por un mensaje para elegir un camino entre los canales de la red. El encaminamiento puede ser visto como un par (R, ρ), donde

$$R \subset C \times N \times C,$$
$$\rho : P(C) \times \alpha \longmapsto C.$$

La relación de encaminamiento R identifica los caminos permitidos que pueden ser usados por un mensaje para alcanzar su destino. Dada la posición actual del mensaje, C, y su nodo destino, N, R identifica un conjunto de canales permitidos, C, que pueden ser usados como siguiente paso en el camino.

La función ρ selecciona uno de los caminos de entre los permitidos. En cada paso del encaminamiento, ρ toma el conjunto de posibles canales siguientes, $P(C)$, alguna información adicional acerca del estado de la red, α, y elige en canal en concreto, C. La información adicional, α, puede ser constante, aleatoria, o basada en información sobre el tráfico de la red.

Los métodos de encaminamiento pueden clasificarse en *deterministas*, *inconscientes* (*oblivious*), o adaptativos. Con el encaminamiento determinista, el camino que sigue un mensaje depende únicamente de los nodos origen y destino. En este caso R es una función y α es constante (no se proporciona información adicional).

En un encaminamiento inconsciente se puede elegir varios caminos a través de la red, pero no se utiliza información acerca del estado de la red para elegir un camino. El mensaje no es consciente del resto de tráfico en la red. Con un encaminamiento de este tipo, R es una relación (pueden existir varios caminos permisibles). Para que el encaminamiento sea inconsciente, α no puede contener información acerca del estado de la red. Puede ser aleatoria, una función de tiempo, o una función del contenido del mensaje.

El caso más general es el encaminamiento adaptativo, donde el router puede usar información acerca del estado de la red. Es este caso, α puede ser cualquier función.

5.6.3. Clasificación de las redes de interconexión según su topología

Entre otros criterios, las redes de interconexión de han clasificado en función de modo de funcionamiento (síncrono o asíncrono), y control de la red (centralizado, descentralizado, o distribuido). Hoy en día, los multicomputadores, multiprocesadores, y NOWs dominan el mercado de los computadores paralelos. Todas estas arquitecturas utilizan redes asíncronas con control distribuido. Por lo tanto, nos centraremos en otros criterios que son más significativos en la actualidad.

La figura 5.40 muestra una clasificación de las redes de interconexión conocidas en cuatro grupos en función principalmente de la topología de la red: redes de medio común, redes directas, redes indirectas, y redes híbridas. Para cada grupo, la figura muestra una jerarquía de clases, indicando alguna implementación real que utiliza dicha topología.

En las *redes de medio compartido*, el medio de transmisión está compartido por todos los dispositivos que tienen posibilidad de comunicación. Un enfoque alternativo consiste en tener enlaces punto a punto que conecten de forma directa cada elemento de comunicación con un subconjunto (normalmente reducido) de otros los dispositivos existentes en la red. En este caso, la comunicación entre elementos de comunicación no vecinos requiere la transmisión de la información a través de varios dispositivos intermedios. A estas redes se les denominan *redes directas*. En vez de conectar de forma directa los elementos de comunicación, las *redes indirectas* los conectan mediante uno o más conmutadores. Si existen varios conmutadores, estos suelen estar conectados entre ellos mediante enlaces punto a punto. En este caso, cualquier comunicación entre distintos dispositivos requiere transmitir la información a través de uno o más conmutadores. Finalmente, es posible una aproximación *híbrida*.

5.6.4. Redes de medio compartido

La estructura de interconexión menos compleja es aquella en la que el medio de transmisión está compartido por todos los elementos de comunicación. En estas *redes de medio compartido*, sólo un dispositivo puede utilizar la red en un momento dado. Cada elemento conectado a la red tiene circuitos para manejar el paso de direcciones y datos. La red en sí misma actúa, normalmente, como un elemento pasivo ya que no genera mensajes.

Un concepto importante es el de *estrategia de arbitraje* que determina cómo se resuelven los conflictos de acceso al medio. Una característica de un medio compartido es la posibilidad de soportar un *broadcast* atómico en donde todos los dispositivos conectados al medio pueden monitorizar actividades y recibir la información que se está transmitiendo en el medio compartido. Esta propiedad es importante para un soporte eficiente de muchas aplicaciones que necesitan comunicaciones del tipo "uno a todos" o "uno a muchos", como las barreras de sincronización y protocolos de coherencia de caché basados en *snoopy*. Debido al limitado ancho de banda, un medio compartido únicamente puede soportar un número limitado de dispositivos antes de convertirse en un cuello de botella.

Las redes de medio compartido pueden dividirse en dos grandes grupos: las redes de área local, usadas principalmente en la construcción de redes de ordenadores cuya distancia máxima no supera unos pocos kilómetros, y los buses usados en la comunicación

- Redes de medio compartido
  - Redes de área local
    - Bus de contención (Ethernet)
    - Bus de tokens (Arenet)
    - Anillo de tokens (FDDI Ring, IBM Token Ring)
  - Bus de sistema (Sun Gigaplane, DEC AlphaServer8X00, SGI PoweerPath-2)
- Redes directas (Redes estáticas basadas en encaminador)
  - Topologías estrictamente ortogonales
    - Malla
      - Malla 2-D (Intel Paragon)
      - Malla 3-D (MIT J-Machine)
    - Toros (n-cubo k-arios)
      - Toro 1-D unidireccional o anillo (KSR forst-level ring)
      - Toro 2-D bidireccional (Intel/CMU iWarp)
      - Toro 2-D bidireccional (Cray T3D, Cray T3E)
    - Hipercubo (Intel iPSC, nCUBE)
  - Otras topologías directas: Árboles, Ciclos cubo-conectados, Red de Bruijn, Grafos en Estrella, etc.
- Redes Indirectas (Redes dinámicas basadas en conmutadores)
  - Topologías Regulares
    - Barra cruzada (Cray X/Y-MP, DEC GIGAswitch, Myrinet)
    - Redes de Interconexión Multietapa (MIN)
      - Redes con bloqueos
        MIN Unidireccionales (NEC Cenju-3, IBM RP3)
        MIN Bidireccional (IBM SP, TMC CM-5, Meiko CS-2)
      - Redes sin bloqueos: Red de Clos
  - Topologías Irregulares (DEC Autonet, Myrinet, ServerNet)
- Redes Híbridas
  - Buses de sistema múltiples (Sun XDBus)
  - Redes jerárquicas (Bridged LANs, KSR)
    - Redes basadas en Agrupaciones (Stanford DASH, HP/Convex Exemplar)
  - Otras Topologías Hipergrafo: Hiperbuses, Hipermallas, etc.

Figura 5.40: Clasificación de las redes de interconexión. (1-D = unidimensional; 2-D = bidimensional; 3-D = tridimensional; CMU = Carnegie Mellon University; DASH = Directory Architecture for Shared-Memory; DEC = Digital Equipment Corp.; FDDI = Fiber Distributed Data Interface; HP = Hewlett-Packard; KSR = Kendall Square Research; MIN = Multistage Interconnection Network; MIT = Massachusetts Institute of Technology; SGI = Silicon Graphics Inc.; TMC = Thinking Machines Corp.)

interna de los uniprocesadores y multiprocesadores.

Bus del sistema (*Backplane bus*)

Un *bus del sistema* es la estructura de interconexión más simple para los multiprocesadores basados en bus. Se usa normalmente para interconectar procesadores y módulos de memoria para proporcionar una arquitectura UMA. La figura 5.41 muestra una red con un bus único. Un bus de este tipo consta usualmente de 50 a 300 hilos físicamente realizados mediante una placa base.

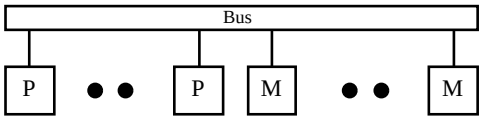


Figura 5.41: Una red con bus único. (M = memoria; P = procesador.)

Existen tres tipos de información en un bus de este tipo: datos, direcciones y señales de control. Las señales de control incluyen la señal de petición del bus y la señal de permiso de acceso al bus, entre muchas otras. Además del ancho de las líneas de datos, el máximo ancho de banda depende de la tecnología. El número de procesadores que pueden conectarse al bus depende de varios factores, como la velocidad del procesador, el ancho de banda del bus, la arquitectura caché y el comportamiento del programa.

LANs de medio compartido

Es posible utilizar una LAN de alta velocidad como una columna vertebral que permita interconectar ordenadores para proporcionar un entorno de computación distribuido. Físicamente, una LAN usa hilos de cobre o fibra óptica como medio de transmisión. La topología utilizada puede ser un bus o un anillo. Debido a razones de implementación y rendimiento, no es práctico tener un control centralizado o algún mecanismo fijo de asignación de acceso que determine quién puede acceder al bus. Las tres alternativas principales de LANs basadas en un control distribuido se describen a continuación.

Bus de contención: El mecanismo de arbitraje de bus más popular es que todos los dispositivos compitan para tener el acceso exclusivo al bus. Debido a la compartición del medio todos los dispositivos pueden monitorizar el estado del bus y detectar colisiones. Aquí, el término "colisión" significa que dos o más dispositivos están usando el bus al mismo tiempo y sus datos colisionan. Cuando se detecta una colisión, los dispositivos causantes de la misma abortan la transmisión para intentarlo posteriormente. Entre las LANs que utilizan este mecanismo está la Ethernet que adopta el protocolo CSMA/CD (*Carrier-Sense Multiple Access with Collision Detection*). El ancho de banda de la Ethernet es 10 Mbps y la distancia máxima es de 250 metros (cable coaxial). Para romper la barrera de 10 Mbps ha aparecido Fast Ethernet que puede proporcionar un ancho de banda de 100 Mbps.

Token Bus: Una desventaja del bus de contención es su naturaleza no determinista, ya que no puede garantizar cuánto se debe esperar hasta ganar el acceso al bus. Por lo tanto, el bus de contención no es el idóneo para soportar aplicaciones de tiempo real. Para eliminar el comportamiento no determinista, aparece un enfoque alternativo que implica pasar un testigo entre los dispositivos conectados a la red. El dispositivo que

tiene el testigo tiene el acceso al bus. Cuando termina de transmitir sus datos, el testigo se pasa al siguiente dispositivo según algún esquema prefijado. Restringiendo el tiempo máximo que se puede estar en posesión del testigo se puede garantizar una cota superior al tiempo que un dispositivo debe esperar. Arcnet soporta token bus con un ancho de banda de 2.4 Mbps.

Token Ring: Una extensión natural al token bus es la de utilizar una estructura de anillo. El token ring de IBM proporciona anchos de banda de 4 y 16 Mbps sobre cable coaxial. El protocolo FDDI (*Fiber Distributed Data Interface*) es capaz de proporcionar un ancho de banda de 100 Mbps usando fibra óptica.

5.6.5. Redes Directas

La escalabilidad es una característica importante en el diseño de sistemas multiprocesador. Los sistemas basados en buses no son escalables al convertirse el bus en un cuello de botella cuando se añaden más procesadores. Las *redes directas* o *redes punto a punto* son una arquitectura red popular y que escalan bien incluso con un número elevado de procesadores. Las redes directas consisten en un conjunto de *nodos*, cada uno directamente conectado a un subconjunto (usualmente pequeño) de otros nodos en la red. En la figura 5.43 se muestra varios tipos de redes directas. Los correspondientes patrones de interconexión se estudiarán posteriormente. Cada nodo es un ordenador programable con su propio procesador, memoria local, y otros dispositivos. Estos nodos pueden tener diferentes capacidades funcionales. Por ejemplo, el conjunto de nodos puede contener procesadores vectoriales, procesadores gráficos, y procesadores de E/S. La figura 5.42 muestra la arquitectura de un nodo genérico. Un componente común en estos nodos es un *encaminador* (router) que se encarga de manejar la comunicación entre los nodos a través del envío y recepción de mensajes. Por esta razón, las redes directas también son conocidas como redes basadas en routers. Cada router tiene conexión directa con el router de sus vecinos. Normalmente, dos nodos vecinos están conectados por un par de canales unidireccionales en direcciones opuestas. También se puede utilizar un canal bidireccional para conectar dos nodos. Aunque la función del encaminador puede ser realizada por el procesador local, los encaminadores dedicados se han usado de forma habitual en multicomputadores de altas prestaciones, permitiendo el solapamiento de la computación y las comunicaciones dentro de cada nodo. Al aumentar el número de nodos en el sistema, el ancho de banda total de comunicaciones, memoria y capacidad de procesamiento del sistema también aumenta. Es por esto que las redes directas son una arquitectura popular para construir computadores paralelos de gran escala.

Cada encaminador soporta un número de canales de entrada y salida. Los canales *internos* o *puertos* conectan el procesador/memoria local al encaminador. Aunque normalmente existe únicamente un par de canales internos, algunos sistemas usan más canales internos para evitar el cuello de botella entre el procesador/memoria local y el router. Los canales *externos* se usan para comunicaciones entre los routers. Conectando los canales de entrada de un nodo a los canales de salida del otros nodos podemos definir la red directa. A no ser que se diga lo contrario, utilizaremos el término "canal" para referirnos a un canal externo. A dos nodos conectados directamente se les llaman *vecinos* o nodos *adyacentes*. Normalmente, cada nodo tiene un número fijo de canales de entrada y salida, cada canal de entrada está emparejado con el correspondiente canal de salida. A través de todas las conexiones entre estos canales, existen varias maneras de conectar los diferentes nodos. Obviamente, cada nodo de la red debe poder alcanzar

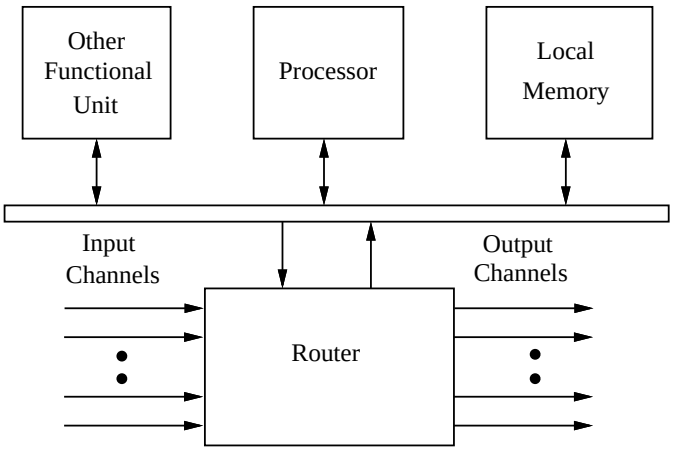


Figura 5.42: Arquitectura de un nodo genérico.

cualquier otro nodo.

5.6.6. Redes Indirectas

Las *redes indirectas* o *basadas en conmutadores* (*switch*) forman el tercer grupo de redes de interconexión. En lugar de proporcionar una conexión directa entre algunos nodos, la comunicación entre cualquier pareja de nodos se realiza a través de *conmutadores*. Cada nodo tiene un adaptador de red que se conecta a un conmutador. Cada conmutador consta de un conjunto de *puertos*. Cada puerto consta de un enlace de entrada y otro de salida. Un conjunto (posiblemente vacío) de puertos en cada conmutador están conectados a los procesadores o permanecen abiertos, mientras que el resto de puertos están conectados a puertos de otros conmutadores para proporcionar conectividad entre los procesadores. La interconexión de estos conmutadores define la topología de la red.

Las redes conmutadas han evolucionado considerablemente con el tiempo. Se han propuesto un amplio rango de topologías, desde topologías regulares usadas en los procesadores array (matriciales) y multiprocesadores de memoria compartida UMA a las topologías irregulares utilizadas en la actualidad en los NOWs. Las topologías regulares tienen patrones de conexión entre los conmutadores regulares mientras que las topologías irregulares no siguen ningún patrón predefinido. En el tema siguiente se estudiarán más detenidamente las topologías regulares. La figura 5.44 muestra una red conmutada típica con topología irregular. Ambos tipos de redes pueden clasificarse además según el número de conmutadores que tiene que atravesar un mensaje para llegar a su destino. Aunque esta clasificación no es importante en el caso de topologías irregulares, puede significar una gran diferencia en el caso de redes regulares ya que algunas propiedades específicas se pueden derivar en función de este dato.

5.6.7. Redes Híbridas

En esta sección describiremos brevemente algunas topologías que no se encuadran en las vistas hasta ahora. En general, las redes híbridas combinan mecanismos de redes de medio compartido y redes directas o indirectas. Por tanto, incrementan el ancho de

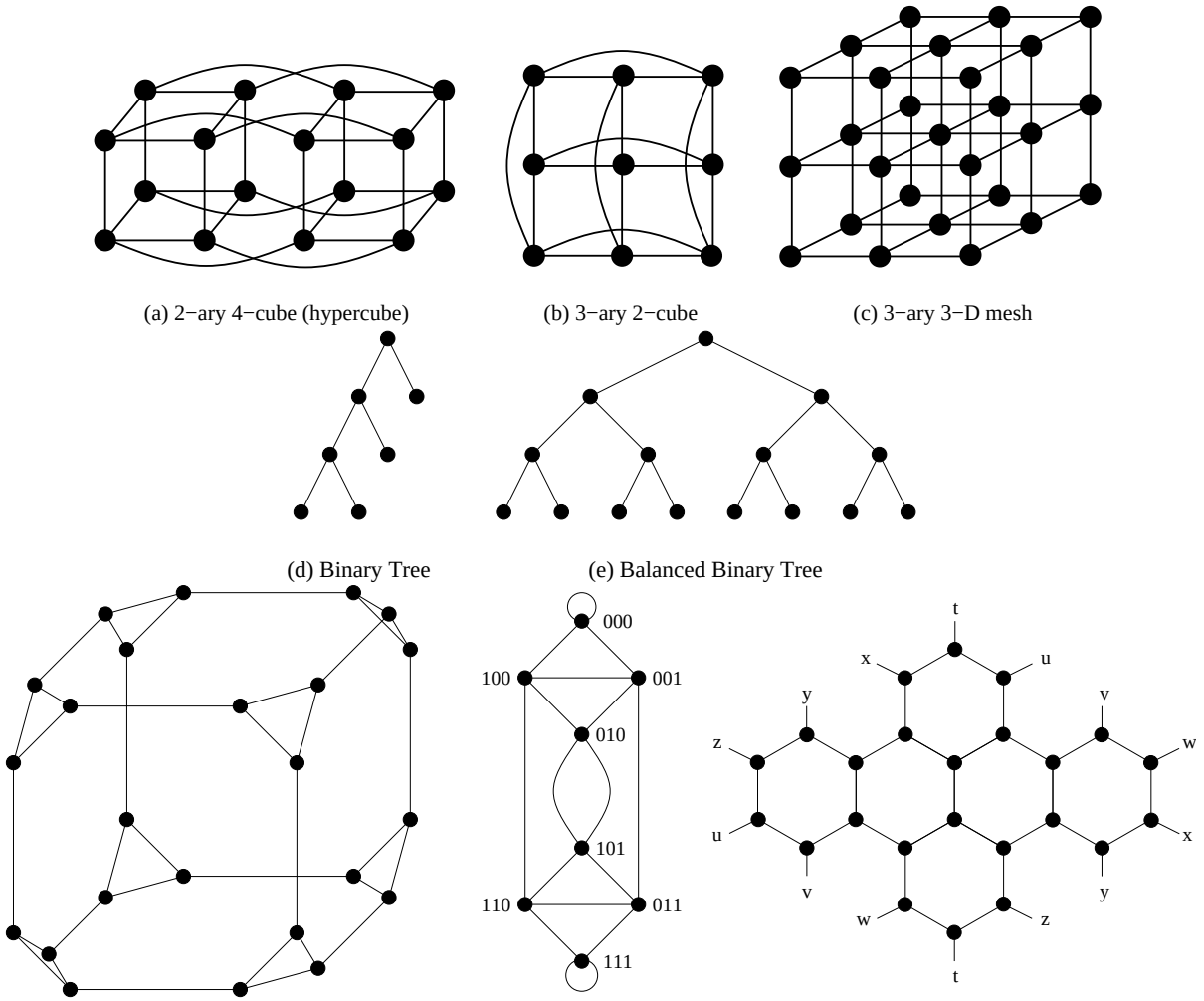


Figura 5.43: Algunas topologías propuestas para redes directas.

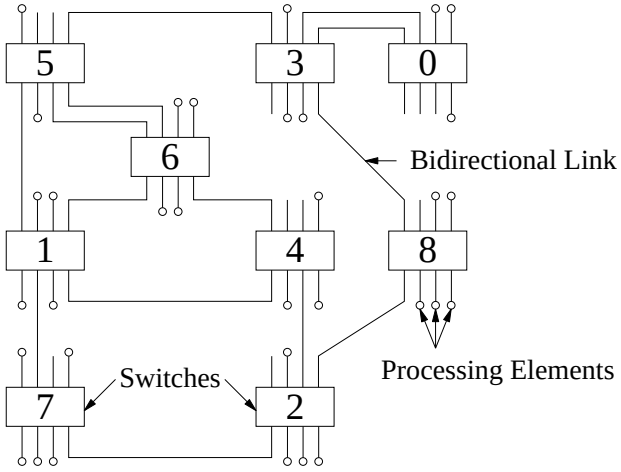


Figura 5.44: Un red conmutada con topología irregular.

banda con respecto a las redes de medio compartido, y reducen la distancia entre nodos con respecto a las redes directas e indirectas. Existen varias aplicaciones concretas de las redes híbridas. Este es el caso de las *bridged LANs*. Sin embargo, para sistemas que necesitan muy alto rendimiento, las redes directas e indirectas consiguen una mejor escalabilidad que las redes híbridas ya que los enlaces punto a punto son más sencillos y más rápidos que los buses de medio compartido. La mayoría de los computadores paralelos de alto rendimiento usan redes directas o indirectas. Recientemente las redes híbridas han ganado aceptación de nuevo. El uso de tecnología óptica permite la implementación de redes de alto rendimiento.

Se han propuesto redes híbridas para diferentes propósitos. En general, las redes híbridas pueden modelarse mediante hipergrafos, donde los vértices del hipergrafo representan un conjunto de nodos de procesamiento, y las aristas representan el conjunto de canales de comunicación y/o buses. Obsérvese que una arista en un hipergrafo puede conectar un número arbitrario de nodos. Cuando una arista conecta exactamente dos nodos entonces representa un canal punto a punto. En caso contrario representa un bus. En algunos diseños de redes, cada bus tiene un único nodo conductor. Ningún otro dispositivo puede utilizar ese bus. En ese caso, no existe necesidad de arbitraje. Sin embargo, todavía es posible la existencia de varios receptores en un tiempo dado, manteniéndose la capacidad de *broadcast* de los buses. Obviamente, cada nodo de la red debe poder controlar al menos un bus, por lo que el número de buses necesarios no puede ser menor que el número de nodos. En este caso, la topología de la red se puede modelar mediante un hipergrafo directo.

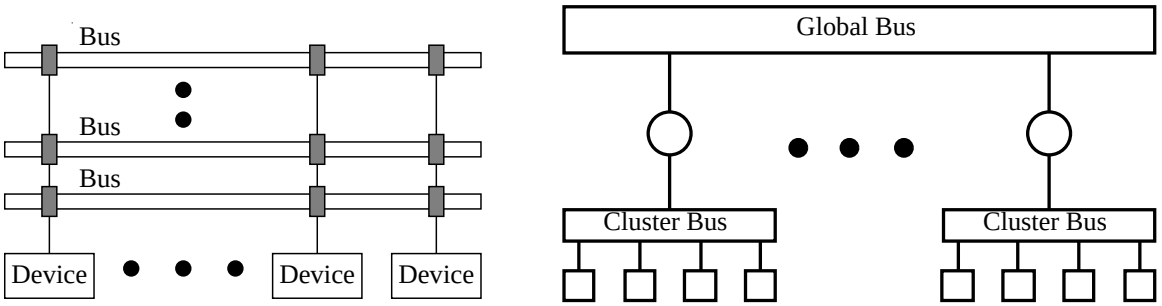


Figura 5.45: Redes Híbridas. (a) Una red multibus. (b) Una jerarquía de dos niveles de buses.

Redes multibus

Debido al ancho de banda limitado que proporcionan las redes de medio compartido, éstas sólo son capaz de soportar un número pequeño de dispositivos, tiene una distancia limitada, y no es escalable. Algunos investigadores han estudiado cómo podría eliminarse ese cuello de botella. Una aproximación para incrementar el ancho de banda de la red se muestra en la figura 5.45a. Sin embargo, los problemas de cableado y el coste del interface hacen que sea de poca utilidad para el diseño de multiprocesadores. Debido a las limitaciones eléctricas de la tecnología del encapsulado, es poco probable tener una red multibus con más de cuatro buses. Sin embargo, utilizando otras tecnologías de empaquetado como la multiplexación por división de la longitud de onda en fibra óptica hace posible integración de múltiples buses.

Redes jerárquicas

Otra aproximación para incrementar el ancho de banda de la red es la que se muestra en la figura 5.45b. Diferentes buses se interconectan mediante routers o puentes para transferir información de un lado a otro de la red. Estos routers o puentes pueden filtrar el tráfico de la red examinando la dirección destino de cada mensaje que le llegue. La red jerárquica permite expandir el área de la red y manejar más dispositivos, pero deja de ser una red de medio compartido. Esta aproximación se usa para interconectar varias LANs. Normalmente, el bus global tiene un mayor ancho de banda. En caso contrario, se convertiría en un cuello de botella. Esto se consigue utilizando una tecnología más rápida. Las redes jerárquicas han sido también propuestas como esquema de interconexión para los multiprocesadores de memoria compartida. También en este caso, el bus global puede convertirse en un cuello de botella.

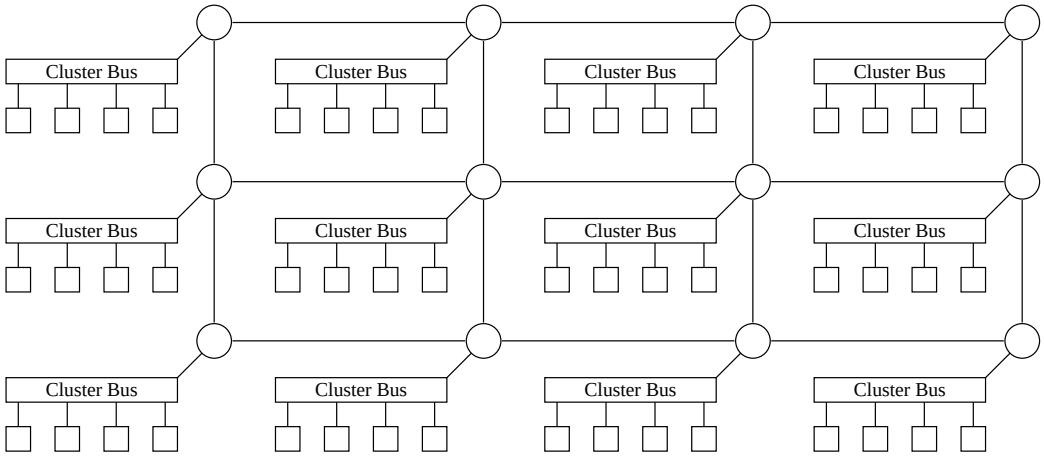


Figura 5.46: Malla bidimensional basada en clusters.

Redes basadas en clusters

Las redes basadas en clusters también tienen una estructura jerárquica. Incluso pueden considerarse como una subclase dentro de las redes jerárquicas. Estas redes combinan las ventajas de dos o más clases de redes a diferentes niveles en la jerarquía. Por ejemplo, es posible combinar las ventajas de los buses y los enlaces punto a punto usando buses en el nivel más bajo de la jerarquía para formar clusters, y una topología directa para conectar los clusters en el nivel superior. Este es el caso del computador paralelo DASH (*Stanford Directory Architecture for Shared-Memory*). La figura 5.46 muestra la arquitectura básica del este computador paralelo. En el nivel inferior cada cluster consta de cuatro procesadores conectados mediante un bus. En el nivel superior, una malla 2-D conecta los clusters. La capacidad *broadcast* del bus se usa a nivel de cluster para implementar un protocolo *snoopy* para mantener la coherencia de la caché. La red directa en el nivel superior elimina la limitación del ancho de banda de un bus, incrementando considerablemente la escalabilidad de la máquina.

Es posible realizar otras combinaciones. En lugar de combinar buses y redes directas, el multiprocesador HP/Convex Exemplar combinan redes directas e indirectas. Este multiprocesador consta de crossbar no bloqueantes de tamaño 5×5 en en nivel inferior de la jerarquía, conectando cuatro bloques funcionales y un interface de E/S

para forma un cluster o *hipernodo.* Cada bloque funcional consta de dos procesadores, dos bancos de memoria e interfaces. Estos hipernodos se conectan a un segundo nivel denominado *interconexión toroidal coherente* formada por múltiples anillos usando el Interface de Coherencia Escalable (SCI). Cada anillo conecta un bloque funcional de todos hipernodos. En el nivel inferior de la jerarquía, el crossbar permite a todos los procesadores dentro del hipernodo acceder a los módulos de memoria entrelazada de ese hipernodo. En el nivel superior, los anillos implementan un protocolo de coherencia de la caché.

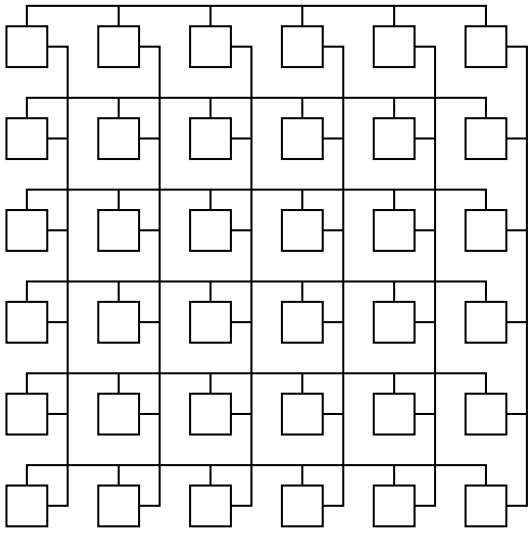


Figura 5.47: Una hipermalla bidimensional.

Otras topologías hipergrafo

Se han propuesto muchas otras topologías híbridas. Entre ellas, una clase particularmente interesante en la de *hipermallas*. Una hipermalla es una topología regular basada en un conjunto de nodos dispuestos en varias dimensiones. En lugar de existir conexiones directas entre los vecinos de cada dimensión, cada nodo está conectado a todos lo nodos de la dimensión a través de un bus. Existen varias formas de implementar una hipermalla. La más directa consiste en conectar todos los nodo de cada dimensión a través de un bus compartido. La figura 5.47 muestra una hipermalla 2-D. En esta red, los buses están dispuestos en dos dimensiones. Cada nodo se conecta a un bus en cada dimensión. Esta topología fue propuesta por Wittie[2], y se le denomina *spanning-bus hypercube*. La misma tiene un diámetro muy pequeño, y la distancia media entre nodos escala muy bien en función del tamaño de la red. Sin embargo, el ancho de banda total no escala bien. Además, los frecuentes cambios en el maestro del bus causan una sobrecarga significativa.

Una implementación alternativa que elimina las restricciones señaladas arriba consiste en reemplazar el bus compartido que conecta los nodos a lo largo de una dimensión dada por un conjunto de tantos buses como nodos existan en esa dimensión. Esta es la aproximación propuesta en la *Distributed Crossbar Switch Hypermesh* (DCSH). La

[2]L. D. Wittie. *Communications structures for large networks of microcomputers.* IEEE Transactions on Computers, vol C-29, pp. 694-702, August 1980.

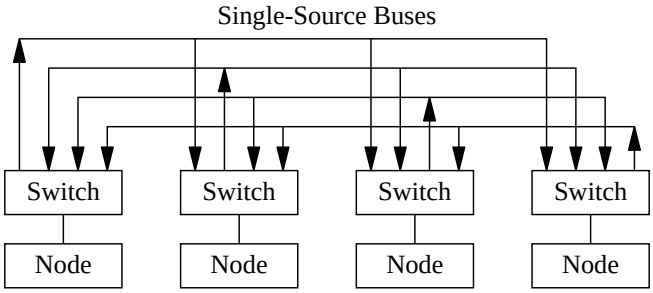


Figura 5.48: Una hipermalla unidimensional con conmutador crossbar distribuido.

figura 5.48 muestra una dimensión de la red. Cada bus es controlado por un único nodo. Por tanto, no hay cambios en la pertenencia del bus. Además, el ancho de banda escala con el número de nodos. Los dos principales problemas, sin embargo, son el alto número de buses necesarios y el alto número de puertos de entrada y salida que necesita cada nodo.

5.7. Redes de interconexión para multiprocesadores

5.7.1. Redes de medio compartido. Buses

Un *bus* de sistema está formado por un conjunto de conductores para la transacción de datos entre procesadores, módulos de memoria, y dispositivos periféricos conectados al bus. En el bus sólo puede haber una transacción a un tiempo entre una fuente (maestro) y uno o varios destinos (esclavos). En el caso de que varios maestros quieran realizar transacciones, la lógica de arbitraje del bus debe decidir quien será el siguiente que consiga el bus y realice la transacción.

Por esta razón a los buses digitales se les llama *buses de contención* o *buses de tiempo compartido*. Un sistema basado en bus tiene un coste bajo comparado con otros sistemas de conexión. Su uso está muy extendido en la industria y existen varios estándares del IEEE disponibles.

El bus es un camino de comunicaciones común entre procesadores, memoria y los subsistemas de entrada/salida. El bus se realiza en la mayoría de los casos sobre una placa de circuito impreso. Las tarjetas con los procesadores, memorias, etc. se conectan a este circuito impreso o placa madre a través de conectores o cables.

El multiprocesador basado en bus es uno de los sistemas multiprocesador más utilizados en computadores de prestaciones medias. Ello es debido a su bajo coste, facilidad de diseño, etc. El principal problema que tienen es su baja escalabilidad lo que no permite tener sistemas con muchos procesadores de forma eficiente. Dependiendo del ancho de banda del bus y de los requisitos de los procesadores que incorpora, un bus puede albergar entre 4 y 16 procesadores de forma eficiente. Por encima de estos números máximos, que dependen del procesador y el bus, el canal de conexión, en este caso el bus, se convierte en el cuello de botella del sistema.

A pesar de esto, y dado que los requisitos de la mayoría de los sistemas informáticos no necesitan de muchos procesadores, se suele utilizar el sistema basado en bus por su bajo coste y facilidad de diseño. Incluso en sistemas más complejos se sigue utilizando

el bus, de manera que para aumentar la escalabilidad del sistema se disponen varios buses formando una jerarquía, o bien, se interconectan varios buses entre sí a través de una red.

Hay varios estándares de buses disponibles para el diseñador. Muchas veces estos buses han surgido a partir de un sistema concreto y se ha intentado luego que fuera lo más estándar posible con los problemas que este tipo de política suele acarrear. Ejemplos de esto los encontramos en el bus del PC, que actualmente es obsoleto, o el popular VME con sus sucesivas extensiones que sigue siendo, a pesar de todo, el bus más utilizado en sistemas empotrados a medida.

Para evitar los problemas del paso del tiempo, y garantizar la portabilidad del bus independientemente del procesador, han aparecido en los últimos tiempos buses que son independientes de una arquitectura específica y que además han sido ideados para hacer frente a los sucesivos avances tecnológicos. Un ejemplo de bus de este tipo es el bus de altas prestaciones Futurebus+, que es el estándar 896 del IEEE. Este bus fue creado por un grupo de trabajo a partir de una hoja en blanco, sin tener en cuenta ningún procesador en particular, y con la intención de definir un bus de muy altas prestaciones que hiciera frente a sucesivos avances tecnológicos. Esta labor no ha sido sencilla y de hecho se ha tardado entre 10 y 15 años, desde que surgió la idea, en tener una definición completa del bus.

5.7.2. Redes indirectas

Para aplicaciones de propósito general es necesario el uso de conexiones dinámicas que puedan soportar todos los patrones de comunicación dependiendo de las demandas del programa. En vez de usar conexiones fijas, se utilizan conmutadores y árbitros en los caminos de conexión para conseguir la conectividad dinámica. Ordenados por coste y rendimiento, las redes dinámicas principales son los buses, las redes de conexión multietapa (MIN *Multistage Interconnection Network*), y las redes barras de conmutadores.

El precio de estas redes es debido al coste de los cables, conmutadores, árbitros, y conectores. El rendimiento viene dado por el ancho de banda de la red, la tasa de transmisión de datos, la latencia de la red, y los patrones de comunicación soportados.

Las redes indirectas se pueden modelar mediante un grafo $G(N, C)$ donde N es el conjunto de conmutadores, y C es el conjunto de enlaces unidireccionales o bidireccionales entre conmutadores. Para el análisis de la mayoría de las propiedades, no es necesario incluir los nodos de procesamiento en el grafo. Este modelo nos permite estudiar algunas propiedades interesantes de la red. Dependiendo de las propiedades que se estén estudiando, un canal bidireccional podrá ser modelado como un línea o como dos arcos en direcciones opuestas (dos canales unidireccionales).

Cada conmutador en una red indirecta puede estar conectado a cero, uno, o más procesadores. Únicamente los conmutadores conectados a algún procesador pueden ser el origen o destino de un mensaje. Además, la transmisión de datos de un nodo a otro requiere atravesar el enlace que une el nodo origen al conmutador, y el enlace entre el último conmutador del camino recorrido por el mensaje y el nodo destino. Por lo tanto, la *distancia entre dos nodos* es la distancia entre los conmutadores que conectan esos nodos más dos. De manera similar, el *diámetro* de la red, definido como la máxima distancia entre dos nodos de la red, es la máxima distancia entre dos conmutadores conectados a algún nodo más dos. Obsérvese que la distancia entre dos nodos conectados

a través de un único conmutador es dos.

Como ya se vio en el capítulo dedicado a las redes, las redes de interconexión, y en el caso que nos ocupa ahora, las redes indirectas, pueden caracterizarse por tres factores: topología, encaminamiento y conmutación. La topología define cómo los conmutadores están interconectados a través de los canales, y puede modelarse con un grafo como el indicado anteriormente. Para una red indirecta con N nodos, la topología ideal conectaría esos nodos a través de un único conmutador de $N \times N$. A dicho conmutador se le conoce con el nombre de *crossbar*. Sin embargo, el número de conexiones físicas de un conmutador está limitado por factores hardware tales como el número de pins disponibles y el la densidad máxima del cableado. Estas dificultades imposibilitan el uso de crossbar en redes de gran tamaño. Como consecuencia, se ha propuesto un gran número de topologías alternativas.

En esta sección nos centraremos en las diferentes topologías existentes de redes indirectas, así como los algoritmos de encaminamiento utilizados en estas redes.

5.7.3. Red de barra cruzada

El mayor ancho de banda y capacidad de interconexión se consigue con la red de barra cruzada. Una red de barra cruzada se puede visualizar como una red de una sola etapa de conmutación. Los conmutadores de cada cruce dan las conexiones dinámicas entre cada par destino-fuente, es decir, cada conmutador de cruce puede ofrecer un camino de conexión dedicado entre un par. Los conmutadores se pueden encender o apagar (*on/off*) desde el programa. Una barra cruzada genérica de conmutadores se muestra en la figura 5.49, donde los elementos V (vertical) y H (horizontal) pueden ser indistintamente procesadores, memorias, etc. Por ejemplo son típicas las configuraciones de procesador con memoria compartida, donde los módulos verticales son todo procesadores y los horizontales memorias, o al revés.

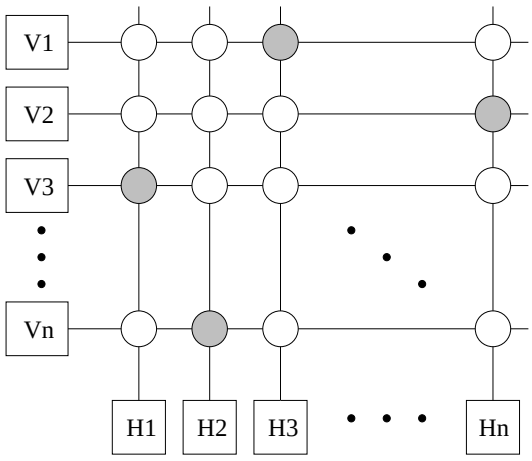


Figura 5.49: La red de conmutación en barra cruzada.

Hay que hace notar, para el caso de multiprocesadores con memoria compartida, que un módulo de memoria sólo puede satisfacer una petición del procesador cada vez. Cuando varias peticiones llegan al mismo módulo de memoria, la propia red debe resolver el conflicto; el comportamiento de cada barra en el conmutador de barra cruzada es muy parecido al de un bus. Sin embargo, cada procesador puede generar una secuencia

de direcciones para acceder a varios módulos de memoria de forma simultánea. Por lo tanto, tal como se muestra en la figura 5.49 suponiendo que los V son procesadores y los H memorias, sólo un conmutador puede activarse en cada columna. Sin embargo, varios conmutadores en diferentes columnas se pueden activar simultáneamente para soportar accesos a memoria paralelos o entrelazados.

Otra aplicación de la red de barra cruzada es la comunicación entre procesadores. La barra cruzada entre procesadores permite conexiones por permutación entre procesadores. Sólo conexiones uno a uno son permitidas, por lo tanto, una barra cruzada $n \times n$ conecta como mucho n pares a un tiempo. Esta es la diferencia con la barra cruzada entre procesadores y memoria. Ambos tipos de barras cruzadas tienen operaciones y propósitos diferentes.

Las redes de barras cruzadas (*crossbar*) permiten que cualquier procesador del sistema se conecte con cualquier otro procesador o unidad de memoria de tal manera que muchos procesadores pueden comunicarse simultáneamente sin contención. Es posible establecer una nueva conexión en cualquier momento siempre que los puertos de entrada y salida solicitados estén libres. Las redes de crossbar se usan en el diseño de multiprocesadores de pequeña escala pero alto rendimiento, en el diseño de *routers* para redes directas, y como componentes básicos en el diseño de redes indirectas de gran escala. Un crossbar se puede definir como una redes conmutada con N entradas y Msalidas, que permite hasta mín$\{N, M\}$ interconexiones punto a punto sin contención. La figura 5.49 muestra un red crossbar de $N \times N$. Aunque lo normal es que N y M sean iguales, no es extraño encontrar redes en las que N y M difieren, especialmente en los crossbar que conectan procesadores y módulos de memoria.

El coste de una red de este tipo es $O(NM)$, lo que hace que sea prohibitiva para valores grandes de N y M. Los crossbars han sido utilizados tradicionalmente en multiprocesadores de memoria compartida de pequeña escala, donde todos los procesadores pueden acceder a la memoria de forma simultánea siempre que cada procesador lea de, o escriba en, un modulo de memoria diferente. Cuando dos o más procesadores compiten por el mismo módulo de memoria, el arbitraje deja proceder a un procesador mientras que el otro espera. El árbitro en un crossbar se distribuye entre todos los puntos de conmutación que conectan a la misma salida. Sin embargo, el esquema de arbitraje puede ser menos complejo que en el caso de un bus, ya que los conflictos en un crossbar son la excepción más que la regla, y por tanto más fáciles de resolver.

Para una red de barras cruzadas con control distribuido, cada punto de conmutación puede estar en uno de los cuatro estados que se muestran en la figura 5.50. En la figura 5.50a, la entrada de la fila en la que se encuentra el punto de conmutación tiene acceso a la correspondiente salida mientras que las entradas de las filas superiores que solicitaban la misma salida están bloqueadas. En la figura 5.50b, a una entrada de una fila superior se le ha permitido el acceso a la salida. La entrada de la fila en la que se encuentra el punto de conmutación no ha solicitado dicha salida, y puede ser propagada a otros conmutadores. En la figura 5.50c, una entrada de una fila superior tiene acceso a la salida. Sin embargo, la entrada de la columna en la que se encuentra en punto de conmutación también ha solicitado esa salida y está bloqueada. La configuración de la figura 5.50(d) sólo es necesaria si el crossbar tiene soporte para comunicaciones *multicast* (uno a muchos).

Los avances en VLSI permiten la integración del hardware para miles de conmutadores en un único chip. Sin embargo, el número de pines de un chip VLSI no puede excederse de algunos centenares, lo que restringe el tamaño del mayor crossbar que

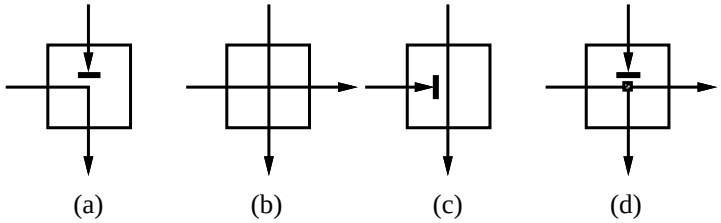


Figura 5.50: Estados de un punto de conmutación en una red de barras cruzadas.

puede integrarse en un único chip VLSI. Crossbars de mayor tamaño se pueden conseguir mediante la partición del mismo en otros de menor tamaño, cada uno de ellos implementado usando un único chip. Por lo tanto, un crossbar de $N \times N$ se puede implementar con $(N/n)(N/n)$crossbar de tamaño $n \times n$.

5.7.4. Redes de interconexión multietapa (MIN)

Los MIN (*Multistage Interconnection Networks* se han utilizado tanto en máquinas MIMD como SIMD. La figura 5.51 muestra una red multietapa generalizada. Un número de $a \times b$ conmutadores se usa en cada etapa. Entre etapas adyacentes se utiliza una red de interconexión fija. Los conmutadores pueden ser programados dinámicamente para establecer las conexiones deseadas entre las entradas y salidas.

Las *redes de interconexión multietapa* (MINs) conectan dispositivos de entrada a dispositivos de salida a través de un conjunto de etapas de conmutadores, donde cada conmutador es una red de barra cruzada. El número de etapas y los patrones de conexión entre etapas determinan la capacidad de encaminamiento de las redes.

Las MINs fueron inicialmente propuestas por las compañías de teléfonos y posteriormente para los procesadores matriciales. En estos casos, un controlador central establece el camino entre la entrada y la salida. En casos en donde el número de entradas es igual al número de salidas, cada entrada puede transmitir de manera síncrona un mensaje a una salida, y cada salida recibir un mensaje de exactamente una entrada. Este patrón de comunicación *unicast* puede representarse mediante una permutación de la dirección asociada a la entrada. Debido a esta aplicación, las MINs se han popularizado como redes de alineamiento que permiten acceder en paralelo a arrays almacenados en bancos de memoria. El almacenamiento del array se divide de tal manera que se permita un acceso sin conflictos, y la red se utiliza para reordenar el array durante el acceso. Estas redes pueden configurarse con un número de entradas mayor que el número de salidas (concentradores) y viceversa (expansores). Por otra parte, en multiprocesadores asíncronos, el control centralizado y el encaminamiento basado en la permutación es inflexible. En este caso, se requiere un algoritmo de encaminamiento que establezca un camino a través de los diferentes estados de la MIN.

Dependiendo del esquema de interconexión empleado entre dos estados adyacentes y el número de estados, se han propuesto varias MINs. Las MINs permiten la construcción de multiprocesadores con centenares de procesadores y han sido utilizadas en algunas máquinas comerciales.

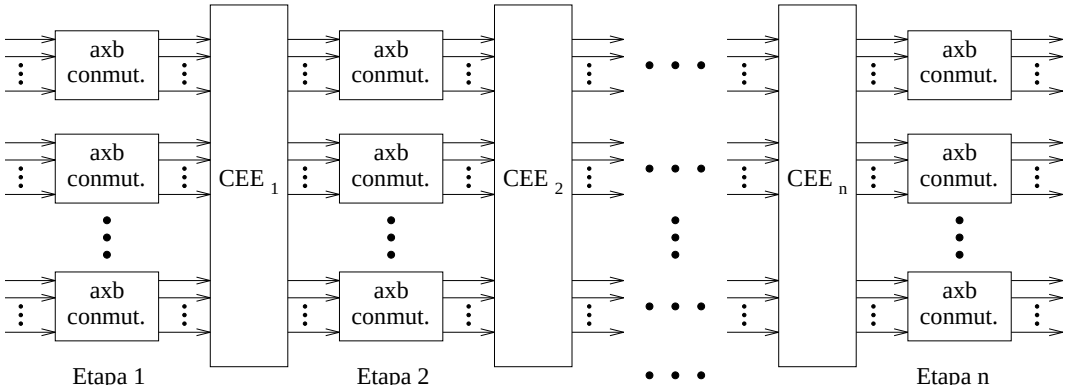


Figura 5.51: Estructura generalizada de una interconexión multietapa (MIN).

Un modelo MIN generalizado

Existen muchas formas de interconectar etapas adyacentes. La figura 5.51 muestra una red de interconexión general con N entradas y M salidas. La red consta de g *etapas*, G_0 a G_{g-1}. Como se muestra en la figura 5.51, cada etapa, G_i, tiene w_i conmutadores de tamaño $a_{i,j} \times b_{i,j}$, donde $1 \leq j \leq w_i$. Así, la etapa $G_i$consta de p_i entradas y q_i salidas, donde

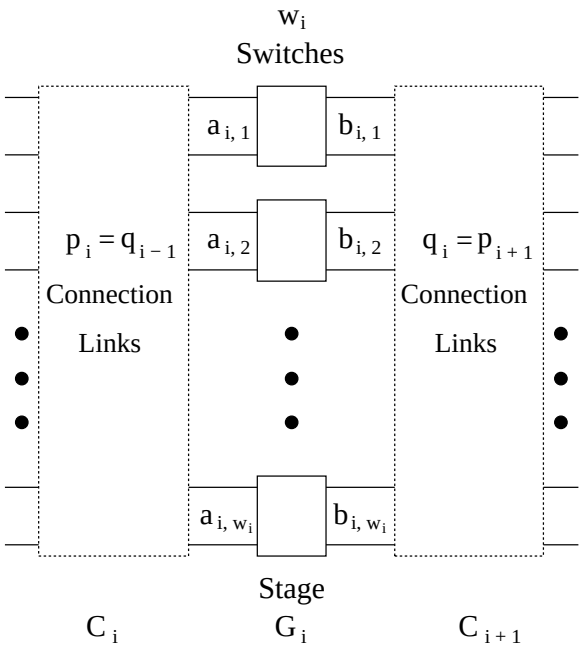


Figura 5.52: Visión detallada de una etapa G_i.

$$p_i = \sum_{j=1}^{w_i} a_{i,j} \quad \text{y} \quad q_i = \sum_{j=1}^{w_i} b_{i,j}$$

La conexión entre dos etapas adyacentes G_{i-1} y G_i, denotada por C_i, define el *patrón de conexión* para $p_i = q_{i-1}$ enlaces, donde $p_0 = N$ y $q_{g-1} = M$. Por lo tanto, una MIN

puede representarse como

$$C_0(N)G_0(w_0)C_1(p_1)G_1(w_1)\ldots G_{g-1}(w_{g-1})C_g(M)$$

Un patrón de conexión $C_i(p_i)$ define cómo deben de estar conectados los enlaces p_i entre las $q_{i-1} = p_i$ salidas de la etapa G_{i-1}y las p_i entradas de la etapa G_i. Patrones diferentes de conexión dan lugar a diferentes características y propiedades topológicas de las MINs. Los enlaces de C_i están numerados de 0 a $p_i - 1$.

Desde un punto de vista práctico, es interesante que todos los conmutadores sean idénticos, para amortizar el coste de diseño. Las redes Banyan son una clase de MINs con la propiedad de que existe un único camino entre cada origen y destino. Una red Delta N-nodo es una subclase de red Banyan, que se construye a partir de $k \times k$ conmutadores en n etapas. donde cada etapa contiene $\frac{N}{k}$ conmutadores. Muchas de las redes multietapa más conocidas, como la Omega, Inversa, Cubo, Mariposa, y de Línea base, pertenecen a la clase de redes Delta, habiéndose demostrado ser topológica y funcionalmente equivalentes.

En el caso de que los conmutadores tengan el mismo número de puertos de entrada y salida, las MINs también tendrán el mismo número de puertos de entrada y salida. Dado que existe una correspondencia uno a uno entre las entradas y las salidas, a estas conexiones se les denominan *permutaciones*. A continuación definiremos cinco permutaciones básicas. Aunque estas permutaciones fueron originariamente definidas para redes con conmutadores 2×2, la mayoría de las definiciones se pueden extender pare redes con conmutadores $k \times k$ y que tienen $N = k^n$ entradas y salidas, donde n es un entero. Sin embargo, algunas de estas permutaciones sólo están definidas para el caso de que N sea potencia de dos. Con $N = k^n$ puertos, sea $X = x_{n-1}x_{n-2}\ldots x_0$ la codificación de un puerto arbitrario, $0 \leq X \leq N-1$, donde $0 \leq x_i \leq k-1$, $0 \leq i \leq n-1$.

5.7.5. Tipos de etapas de permutación para MIN

Las diferentes clases de redes multietapa se diferencian en los módulos conmutadores empleados y en los patrones de la *conexión entre etapas* (CEE). El módulo conmutador más simple es el 2×2, y los patrones para la CEE más usados suelen ser el barajado perfecto, mariposa, barajado multivía, barra cruzada, conexión cubo, etc. Veamos a continuación algunos de estos patrones fijos de interconexión.

Conexión de barajado perfecto

El patrón de barajado perfecto tiene un amplio campo de aplicación en las interconexiones multietapa. Fue originalmente propuesto para calcular la transformada rápida de Fourier. La permutación entre los elementos de entrada y salida de la red está basada en la mezcla perfecta de dos montones de cartas iguales que consiste en intercalar una a una las cartas de un montón con las del otro montón. La red de barajado perfecto toma la primera mitad de las entradas y la entremezcla con la segunda mitad, de manera que la primera mitad pasa a las posiciones pares de las salidas, y la segunda mitad a las impares.

La permutación *k-baraje perfecto*, σ^k, se define por

$$\sigma^k(X) = (kX + \left\lfloor \frac{kX}{N} \right\rfloor) \bmod N$$

Un modo más sólido de describir dicha conexión es

$$\sigma^k(x_{n-1}x_{n-2}\dots x_1x_0) = x_{n-2}\dots x_1x_0x_{n-1}$$

La conexión *k-baraje perfecto* realiza un desplazamiento cíclico hacia la izquierda de los dígitos de X en una posición. Para $k = 2$, esta acción se corresponde con el barajado perfecto de una baraja de N cartas, como se demuestra en la figura 5.53a para el caso de $N = 8$. El baraje perfecto corta la baraja en dos partes y las entremezcla empezando con la segunda parte. El *baraje perfecto inverso* realiza la acción contraria como se define a continuación:

$$\sigma^{k^{-1}}(x_{n-1}x_{n-2}\dots x_1x_0) = x_0x_{n-1}\dots x_2x_1$$

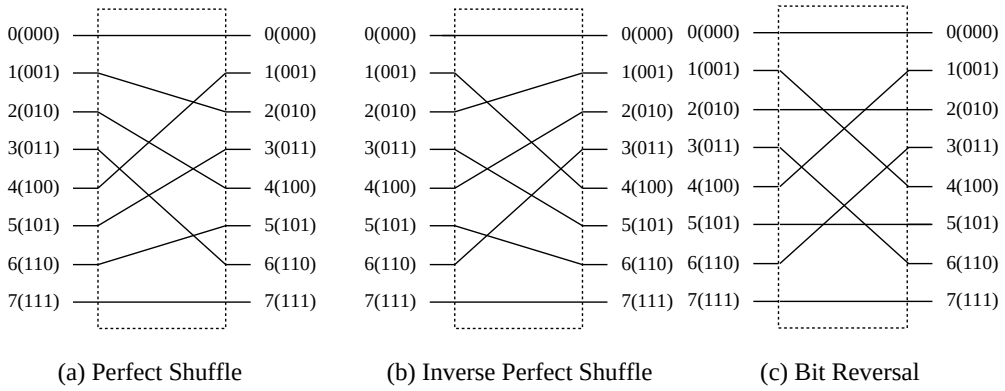


Figura 5.53: Barajado perfecto, barajado perfecto inverso, y bit reversal para $N = 8$.

Suponiendo que las entradas tienen la forma $a_{n-1}a_{n-2}\dots a_1a_0$, donde a_i es el bit i-ésimo de la dirección de cada entrada, el barajado perfecto realiza la siguiente transformación en la dirección:

$$\text{Baraja}(a_{n-1}a_{n-2}\dots a_1a_0) = a_{n-2}a_{n-3}\dots a_1a_0a_{n-1} \tag{5.1}$$

es decir, los bits de la dirección son desplazados cíclicamente una posición hacia la izquierda. La inversa del barajado perfecto los mueve cíclicamente hacia la derecha.

Conexión de dígito inverso (*Digit Reversal Connection*)

La permutación de *dígito inverso* ρ_k está definida por

$$\rho^k(x_{n-1}x_{n-2}\dots x_1x_0) = x_0x_1\dots x_{n-2}x_{n-1}$$

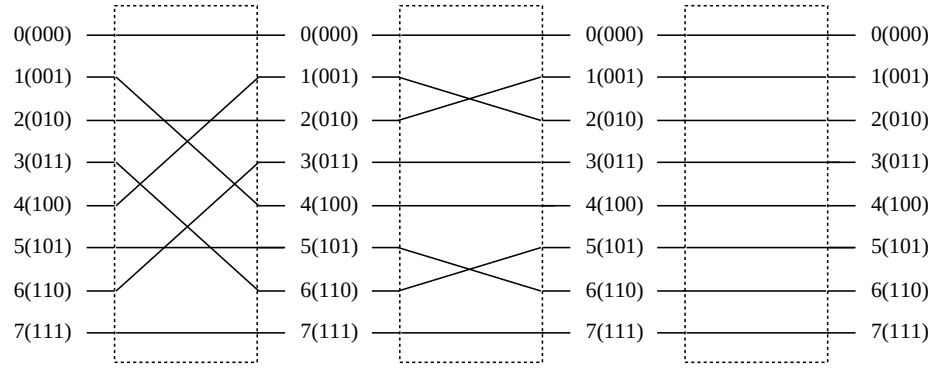
A esta permutación se le suele denominar *bit inverso*, indicando claramente que fue propuesta para $k = 2$. Sin embargo, la definición es también válida para $k > 2$. La figura 5.53c muestra una conexión de bit inverso para el caso $k = 2$ y $N = 8$.

Conexión mariposa

La i-ésima k-aria permutación mariposa β_i^k, para $0 \le i \le n-1$, se define como

$$\beta_i^k(x_{n-1}\dots x_{i+1}x_ix_{i-1}\dots x_1x_0) = x_{n-1}\dots x_{i+1}x_0x_{i-1}\dots x_1x_i$$

La i-ésima permutación mariposa intercambia los dígitos 0 e i-ésimo del índice. La figura 5.54 muestra la conexión mariposa para $k = 2$, $i = 0, 1, 2$ y $N = 8$. Observar que β_0^k define una conexión uno a uno denominada conexión identidad, I.



(a) Second Butterfly (b) First Butterfly (c) Zeroth Butterfly

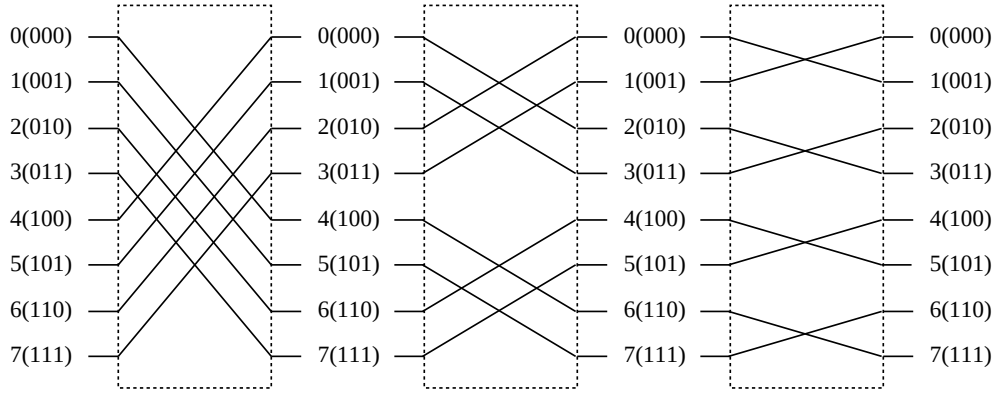
Figura 5.54: La conexión mariposa para $N = 8$.

Conexión Cubo

La i-ésima conexión *cubo* E_i, para $0 \leq i \leq n-1$, se define únicamente para $k = 2$, por

$$E_i(x_{n-1} \ldots x_{i+1} x_i x_{i-1} \ldots x_0) = x_{n-1} \ldots x_{i+1} \overline{x}_i x_{i-1} \ldots x_0$$

La i-ésima conexión cubo complementa el i-ésimo bit del índice. La figura 5.55 muestra la conexión cubo para $i = 0, 1, 2$ y $N = 8$. A E_0 también se le denomina conexión *intercambio*.



(a) Second Cube (b) First Cube (c) Zeroth Cube

Figura 5.55: La conexión cubo para $N = 8$.

Conexión en Línea Base

La i-ésima k-aria permutación en línea base δ_i^k, para $0 \leq i \leq n-1$, se define por

$$\delta_i^k(x_{n-1} \ldots x_{i+1} x_i x_{i-1} \ldots x_1 x_0) = x_{n-1} \ldots x_{i+1} x_0 x_i x_{i-1} \ldots x_1$$

La i-ésima conexión de línea base realiza un desplazamiento cíclico del los $i+1$ dígitos menos significativos del índice una posición a la derecha. La figura 5.56 muestra una conexión en línea base para $k = 2$, $i = 0, 1, 2$ y $N = 8$. Observar que δ_0^k define la conexión identidad I.

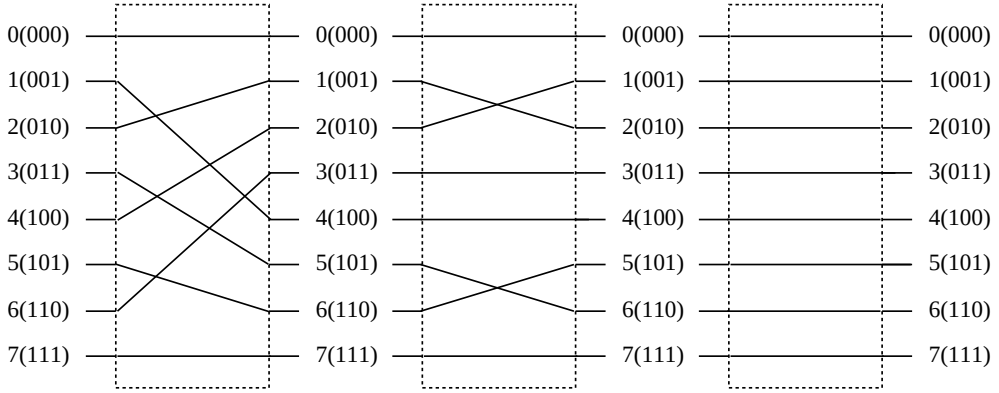


Figura 5.56: La conexión en línea base para $N = 8$.

5.7.6. Clasificación de las redes MIN

Dependiendo de la disponibilidad de caminos para establecer nuevas conexiones, las redes multietapa se han dividido tradicionalmente en tres clases:

1. *Bloqueantes.* La conexión entre un par entrada/salida libre no siempre es posible debido a conflictos entre las conexiones existentes. Típicamente, existe un único camino entre cada par entrada/salida, lo que minimiza el numero de conmutadores y los estados. Sin embargo, es posible proporcionar múltiples caminos para reducir los conflictos e incrementar la tolerancia a fallos. A estas redes bloqueantes también se les denominan *multicamino.*
2. *No bloqueantes.* Cualquier puerto de entrada puede conectarse a cualquier puerto de salida libre sin afectar a las conexiones ya existentes. Las redes no bloqueantes tienen la misma funcionalidad que un crossbar. Estar redes requieren que existan múltiples caminos entre cada entrada y salida, lo que lleva a etapas adicionales.
3. *Reconfigurables.* Cada puerto de entrada puede ser conectado a cada puerto de salida. Sin embargo, las conexiones existentes pueden requerir de un reajuste en sus caminos. Estas redes también necesitan de la existencia de múltiples caminos entre cada entrada y salida pero el numero de caminos y el coste es menor que en el caso de redes no bloqueantes.

La redes no bloqueantes son caras. Aunque son más baratas que un crossbar del mismo tamaño, su costo es prohibitivo para tamaños grandes. El ejemplo más conocido de red no bloqueante multietapa es la red Clos, inicialmente propuesta para redes telefónicas. Las redes reconfigurables requieren menos estados o conmutadores más sencillos que una red no bloqueante. El mejor ejemplo de red reconfigurable es la red Beneš. La figura 5.57 muestra de una Beneš de 8×8. Para 2^n entradas, esta red necesita $2n - 1$ etapas, y proporciona 2^{n-1} caminos alternativos. Las redes reconfigurables requieren un controlador central para reconfigurar las conexiones, y fueron propuestas para los procesadores matriciales (*array processors*). Sin embargo, las conexiones no se pueden reconfigurar fácilmente en multiprocesadores, ya que el acceso de los procesadores a la red es asíncrono. Así, las redes reconfigurables se comportan como redes bloqueantes cuando los accesos son asíncronos. Nos centraremos en la redes bloqueantes.

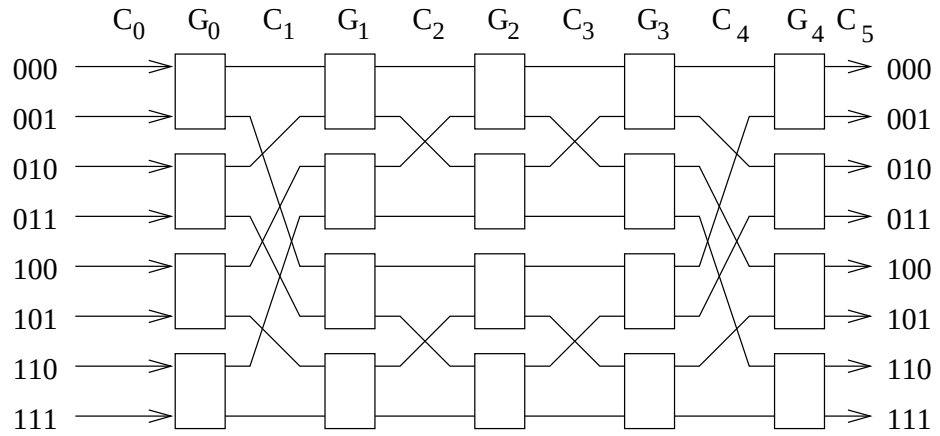


Figura 5.57: Una red Beneš 8×8.

Dependiendo del tipo de canales y conmutadores, las redes multietapa pueden dividirse en dos clases:

1. *MINs unidireccionales*. Los canales y conmutadores son unidireccionales.
2. *MINs bidireccionales*. Los canales y conmutadores son bidireccionales. Esto implica que la información se puede transmitir simultáneamente en direcciones opuestas entre conmutadores vecinos.

Redes de Interconexión Multietapa Unidireccionales

Los bloques básicos de construcción de una red multietapa unidireccional son los conmutadores unidireccionales. Un conmutador $a \times b$ es un *crossbar* con a entradas y b salidas. Si cada puerto de entrada puede conectarse a exactamente un puerto de salida, a lo más existirán mín$\{a, b\}$ conexiones simultáneas. Si cada puerto de entrada puede conectarse a varios puertos de salida, se necesita un diseño más complicado para soportar la comunicación *multicast (*uno a varios). En el modo de comunicación *broadcast* (uno a todos), cada puerto puede conectarse a todos los puertos de salida. La figura 5.58 muestra los cuatro posibles estados de un conmutador 2×2. Los últimos dos estados se usan para soportar las comunicaciones uno a muchos y uno a todos.

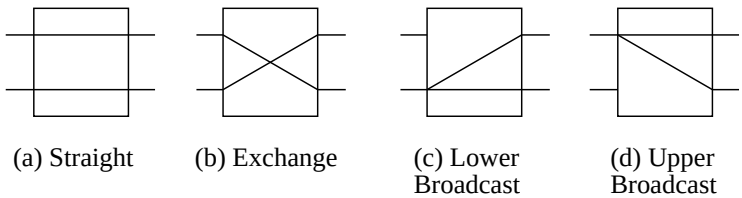
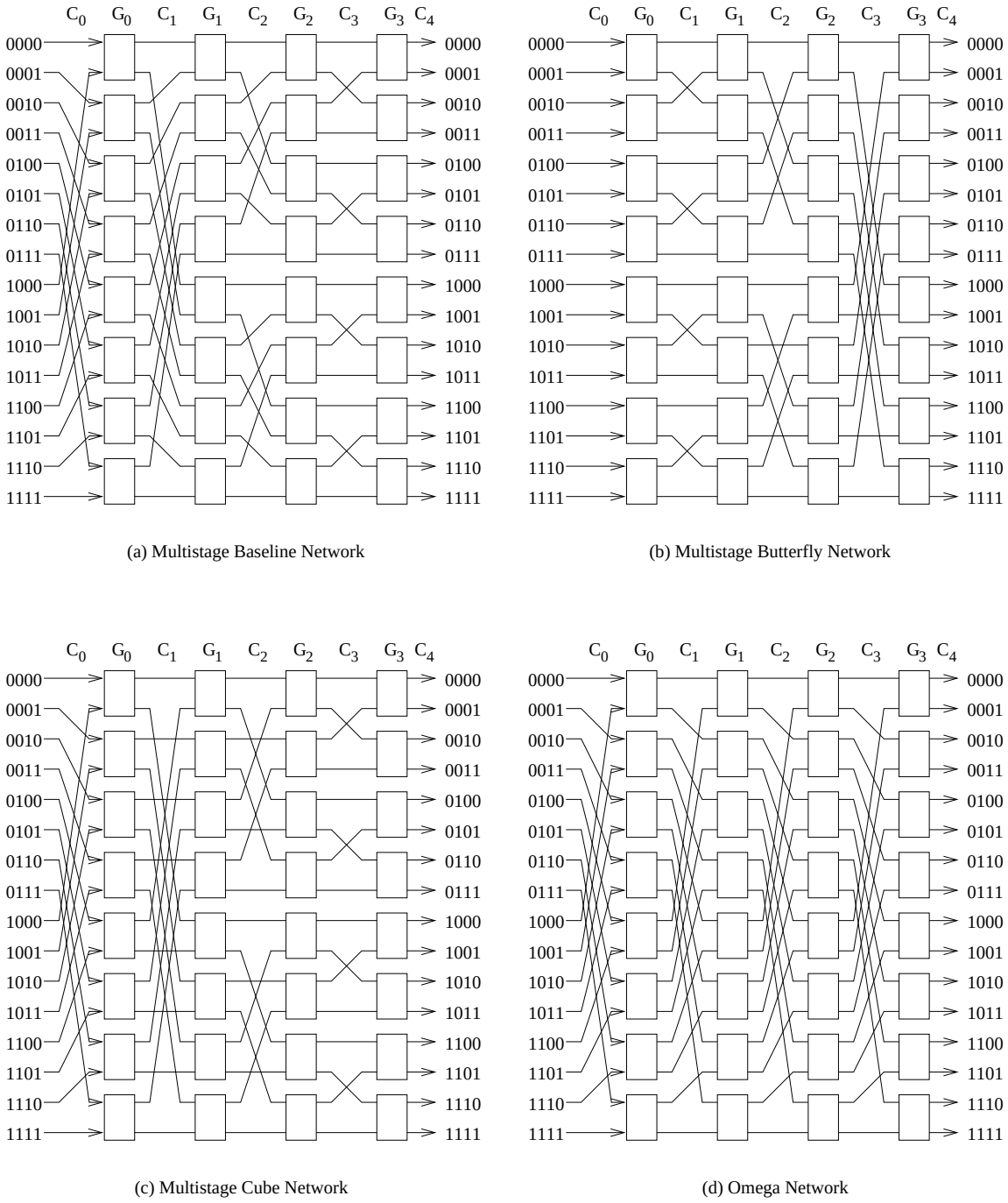


Figura 5.58: Cuatro posibles estados de un conmutador 2×2.

En las redes multietapa con $N = M$, es habitual usar conmutadores con el mismo número de puertos de entrada y de salida ($a = b$). Si $N > M$, se usarán conmutadores con $a > b$. A estos conmutadores también se les denominan *conmutadores de concentración*. En el caso de $N < M$ se usarán *conmutadores de distribución* con $a < b$.

Se puede demostrar que con N puertos de entrada y salida, una red multietapa unidireccional con conmutadores $k \times k$ necesita al menos $\lceil \log_k N \rceil$ etapas para que



(a) Multistage Baseline Network

(b) Multistage Butterfly Network

(c) Multistage Cube Network

(d) Omega Network

Figura 5.59: Cuatro redes de interconexión unidireccionales multietapa de 16×16.

exista un camino entre cualquier puerto de entrada y de salida. Añadiendo etapas adicionales es posible utilizar caminos alternativos para enviar un mensaje desde un puerto de entrada a un puerto de salida. Cualquier camino a través de la MIN cruza todos las etapas. Por tanto, todos los caminos tienen la misma longitud.

A continuación se consideran cuatro topologías equivalentes en redes multietapa unidireccionales. Estas redes pertenecen a la familia de redes Delta.

- **Redes multietapa de línea base** (*Baseline MINs*). En una red multietapa de línea base, el patrón de conexión C_i viene descrito por la $(n-i)$-ésima permutación en línea base δ_{n-i}^k para $1 \leq i \leq n$. El patrón de conexión para C_o viene dado por σ^k.

  La *red de línea base* es un ejemplo de red con una forma muy conveniente de algoritmo de autoencaminamiento, en el que los bits sucesivos de la dirección de destino controlan las sucesivas etapas de la red. Cada etapa de la red divide el rango de encaminamiento en dos. La figura 5.60 muestra este tipo de red suponiendo 4 entradas y 4 salidas. El funcionamiento consiste en que la primera etapa divide el camino en dos vías, uno hacia la parte baja de la red, y el otro hacia la parte alta. Por tanto, el bit más significativo de la dirección de destino puede usarse para dirigir la entrada hacia la mitad alta de la segunda etapa si el bit es 0, o hacia la mitad baja si el bit es 1. La segunda etapa divide la ruta entre el cuarto más alto o el segundo cuarto si la mitad más alta se había seleccionado, o el cuarto más bajo o el primero si la mitad seleccionada era la baja. El segundo bit más significativo es usado para decidir qué cuarto elegir. Este proceso se repite para todas las etapas que haya. Es evidente que el número de bits necesarios para la dirección de las entradas dará el número de etapas necesario para conectar completamente la entrada con la salida. Por último, el bit menos significativo controla la última etapa. Este tipo de redes autorutadas sugiere la transmisión por conmutación de paquetes.

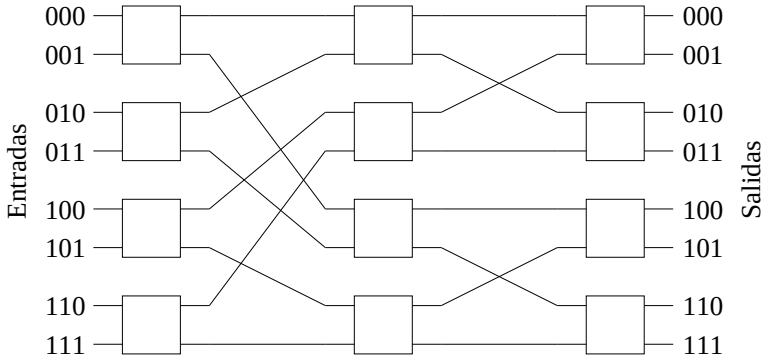
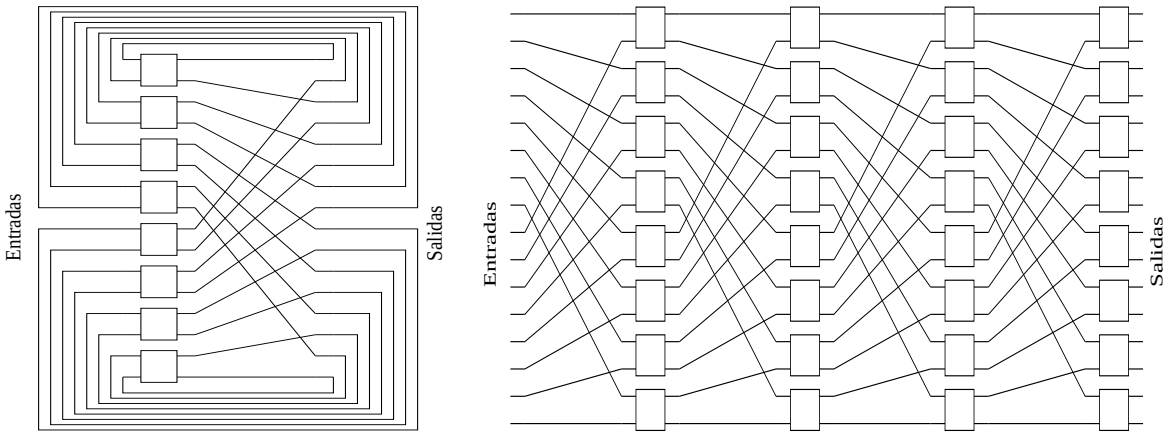


Figura 5.60: Red de línea base 8×8.

  Es fácil ver que una red de este tipo se genera de forma recursiva, ya que se van dividiendo las posibles rutas en mitades donde cada mitad es a su vez una red de línea base y así sucesivamente hasta la última etapa. Los conmutadores son 2×2 con dos posibles estados, directo y cruzado, entre las entradas y salidas.
- **Redes multietapa mariposa** (*Butterfly MINs*). En una red multietapa mariposa, el patrón de conexión $C_i$viene descrito por la i-ésima permutación mariposa β_i^k para $0 \leq i \leq n-1$. El patrón de conexión de C_n es β_0^k.
- **Redes multietapa cubo** (*Cube MINs*). En una red multietapa cubo, el patrón de conexión C_i viene descrito por la $(n-i)$-ésima permutación mariposa β_{n-i}^k para

$1 \leq i \leq n$. El patrón de conexión de C_o es σ^k.

- **Red Omega**. En una red omega, el patrón de conexión C_i viene descrito por la permutación k-baraje perfecto σ^k para $0 \leq i \leq n-1$. El patrón de conexión de C_n es es β_0^k. Así, todos los patrones de conexión menos el último son idénticos. El último patrón de conexión no produce una permutación.
  La figura 5.61(b) muestra la red Omega para 16 elementos de entrada y 16 de salida. Tal y como se comentó anteriormente, son necesarios $\log_2 a$ etapas siendo a el número de entradas a la red. Como cada etapa necesita $n/2$ conmutadores, la red total requiere $n(\log_2 n)/2$ conmutadores, cada uno controlado de forma individual.
  La red Omega se propuso para el procesamiento de matrices utilizando conmutadores de cuatro estados (directo, intercambio, el de arriba a todos, y el de abajo a todos) y actualmente se utiliza en sistemas multiprocesadores.
- **Red de barajado/intercambio**. Es una red un tanto peculiar basada en la red omega. Para hacer todas las posibles interconexiones con el patrón de barajado, se puede utilizar una red de recirculación que devuelve las salidas de nuevo a las entradas hasta que se consigue la interconexión deseada. Para poder hacer esto se añaden *cajas de intercambio* que no son más que conmutadores 2×2. La figura 5.61(a) muestra una red de este tipo para 16 elementos de entrada y 16 de salida. Si hay 2^n entradas es necesario un máximo de n ciclos para realizar cualquier permutación entre los elementos de entrada y salida.



(a) Red de barajado-intercambio (b) Red Omega

Figura 5.61: Redes basadas en el barajado perfecto.

La equivalencia topológica entre estas redes multietapa puede ser vista como sigue: Consideremos que cada enlace de entrada del primer estado se numera usando una cadena de n dígitos $s_{n-1} \ldots s_0$, donde $0 \leq s_i \leq k-1$ para $0 \leq i \leq n-1$. El dígito menos significativo s_0 da la dirección del puerto de entrada en el correspondiente conmutador y la dirección del conmutador viene dada por $s_{n-1}s_{n-2} \ldots s_1$. En cada etapa, un conmutador dado es capaz de conectar cualquier puerto de entrada con cualquier puerto de salida. Esto puede interpretarse como el cambio del valor del dígito menos significativo de la dirección. Para conseguir conectar cualquier entrada con cualquier salida de la red, deberíamos de ser capaces de cambiar el valor de todos los dígitos. Como cada conmutador sólo es capaz de cambiar el dígito menos significativo de la dirección, los

patrones de conexión entre las etapas se definen de tal forma que se permute la posición de los dígitos, y después de n etapas todos los dígitos habrán ocupado la posición menos significativa. Por lo tanto, la MINs definidas arriba difieren en el orden en el que los dígitos de la dirección ocupan la posición menos significativa.

La figura 5.59 muestra la topología de cuatro redes de interconexión multietapa: (a) red *baseline*, (b) red *butterfly*, (c) red *cube*, y (d) red *omega*.

Redes de interconexión multietapa bidireccionales

La figura 5.62 ilustra un conmutador bidireccional en donde cada puerto está asociado a un par de canales unidireccionales en direcciones opuestas. Esto implica que la información entre conmutadores vecinos puede transmitirse simultáneamente en direcciones opuestas. Para facilitar la explicación, supondremos que los nodos procesadores están en la parte izquierda de la red, como se muestra en la figura 5.63. Un conmutador bidireccional soporta tres tipos de conexiones: *hacia delante*, *hacia atrás* y *de vuelta* (ver figura 5.62). Dado que son posibles la conexiones de vuelta entre puertos del mismo lado del conmutador, los caminos tienen diferentes longitudes. En la figura 5.63 se muestra una MIN mariposa bidireccional de ocho nodos.

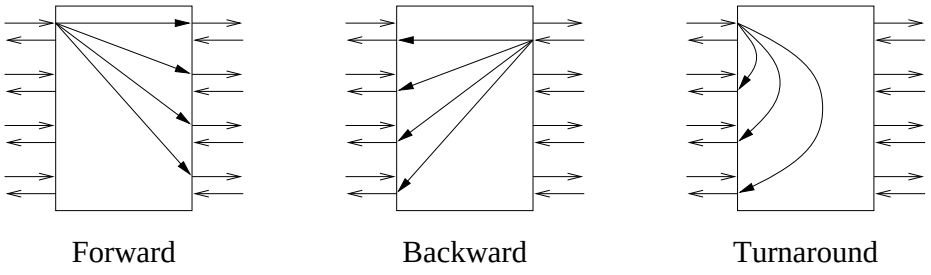


Figura 5.62: Conexiones en un conmutador bidireccional.

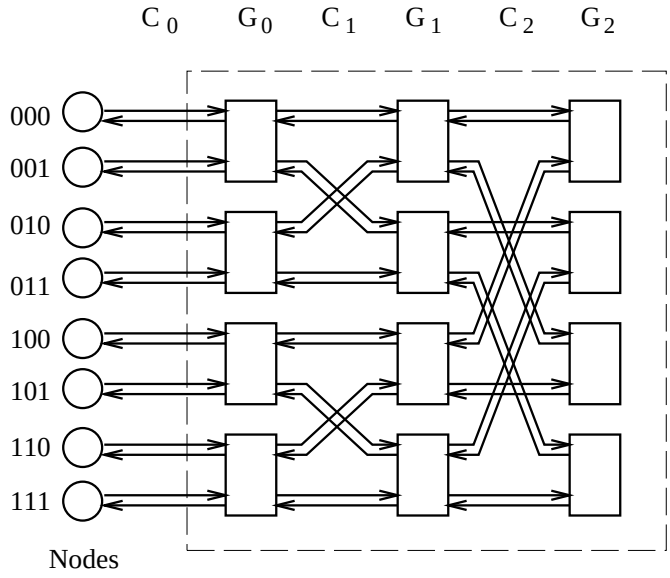


Figura 5.63: Una MIN mariposa bidireccional de ocho nodos.

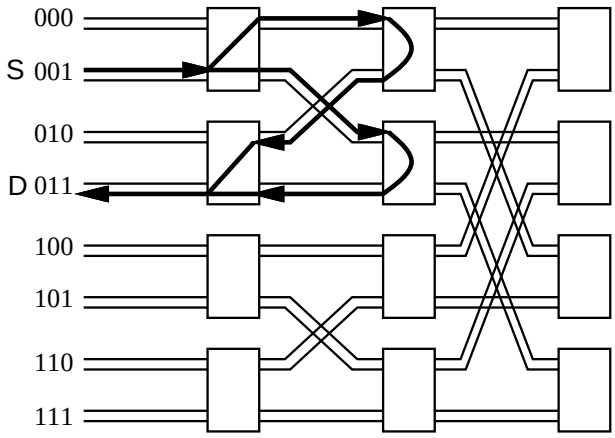


Figura 5.64: Caminos alternativos para una MIN mariposa bidireccional de ocho nodos.

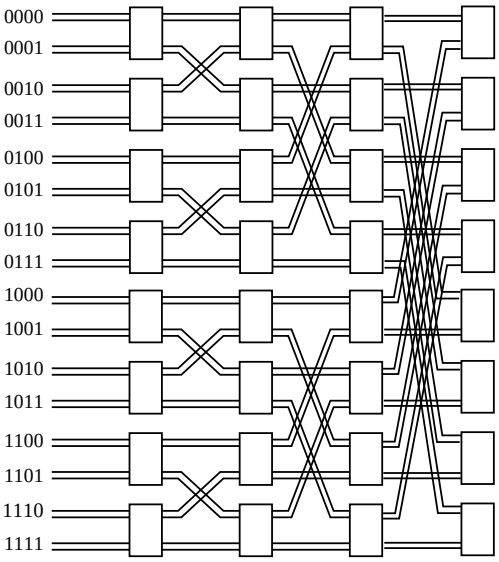
En las BMINs, los caminos se establecen cruzando etapas hacia delante, después estableciendo una conexión de vuelta, y finalmente cruzando los estados en dirección hacia atrás. A este método se le denomina *encaminamiento de vuelta* (*turnaround routing*). La figura 5.64 muestra dos caminos alternativos desde el nodo S al nodo D en una BMIN mariposa de ocho nodos. Cuando se cruzan las etapas hacia delante, existen varios caminos. En cada conmutador se puede seleccionar cualquiera de sus puertos de salida. Sin embargo, una vez cruzada la conexión de vuelta, existe un único camino hasta el nodo destino. En el peor caso, el establecimiento de un camino en una BMIN de n etapas requiere cruzar $2n - 1$ etapas. Este comportamiento es muy parecido al de las redes Beneš. En efecto, la BMIN línea base puede considerarse como una red Beneš plegada.

En la figura 5.65, se muestra un BMIN mariposa con encaminamiento de vuelta. En un *fat tree*, los procesadores se localizan en las hojas, y los vértices internos son conmutadores. El ancho de banda de la transmisión entre los conmutadores se incrementa añadiendo más enlaces en paralelo en los conmutadores cercanos al conmutador raíz. Cuando se encamina un mensaje desde un procesador a otro, se envía hacia arriba en el árbol hasta alcanzar el menor antecesor común de ambos procesadores, y después se envía hacia abajo hacia su destino.

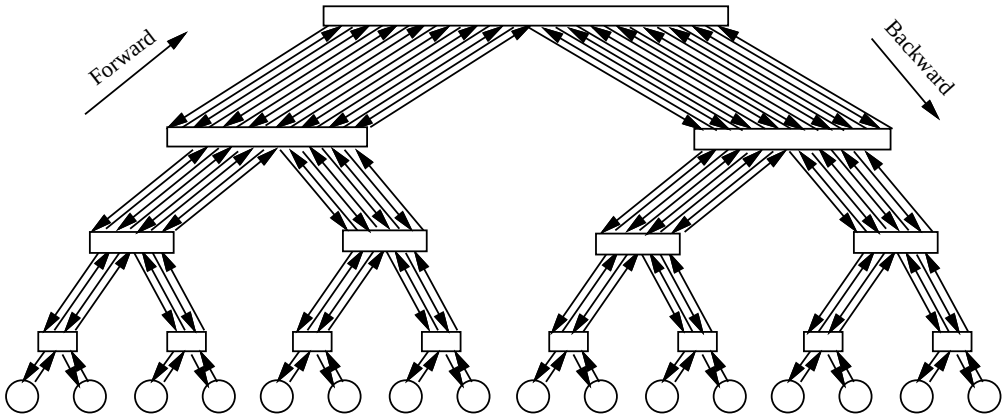
5.7.7. Encaminamiento en redes MIN

En esta sección describiremos diversos aspectos del encaminamiento en redes MINs. Para procesadores matriciales, un controlador central establece el camino entre la entrada y la salida. En casos donde el número de entradas es igual al número de salidas, cada entrada transmite de forma síncrona un mensaje a una salida, y cada salida recibe exactamente un mensaje de una entrada. Calcular las conexiones del conmutador para realizar dicha permutación es una tarea compleja. Además, algunas permutaciones pueden no ser posibles en un paso a través de la red. En este caso, serán necesarios múltiples pasos de los datos a través de la red en cuyo caso el objetivo pasa a ser el minimizar el número de pasadas. La complejidad del cálculo de las conexiones del conmutador es proporcional al número de conmutadores.

Por otra parte, en multiprocesadores asíncronos, el control centralizado y el enca-



(a) A 16−Node Butterfly BMIN Built with 2 x 2 Switches



(b) A 16−Node Fat Tree

Figura 5.65: Árbol grueso y BMIN mariposa.

minamiento por permutación es poco flexible. En este caso es necesario un algoritmo de encaminamiento para establecer el camino por las diferentes etapas de la MIN. La solución más simple consiste en utilizar un encaminamiento en origen. En este caso, el nodo origen especifica el camino completo. Dado que esta solución produce una considerable sobrecarga, nos centraremos en encaminamientos distribuidos. Los algoritmos de encaminamiento para MINs se describirán posteriormente.

Condición de bloqueo en MINs

En este apartado obtendremos condiciones necesarias y suficientes para que se bloqueen dos circuitos, es decir, necesiten el mismo enlace intermedio. En un sistema con N procesadores, existen exactamente N enlaces entre cada etapa de los $k \times k$ conmutadores. La red consta de $n = log_k N$, donde cada etapa está compuesta de $\frac{N}{k}$ conmutadores. Los patrones de enlaces intermedios conectan una salida de un conmutador en la etapa i con un entrada de un conmutador de la etapa $i+1$. El bloqueo ocurre cuando dos paquetes deben atravesar la misma salida en un conmutador de la etapa i, $i = 0, 1, \ldots n-1$.En cada etapa i, podemos numerar las salidas de dicha etapa desde 0 hasta $N-1$. A estas salidas se les denominan salidas intermedias en la etapa i. Si pensamos en la red como una caja negra, podemos representarla como se muestra en la figura 5.66 para el caso de una red Omega con conmutadores 2×2. Cada etapa de conmutadores y cada etapa de enlaces puede representarse como una función que permuta las entradas. En la figura se muestra un ejemplo de un camino desde la entrada 6 a la salida 1. Las salidas intermedias para este camino también se han marcado. Estas son las salidas 4, 0 y 1 para los conmutadores de las etapas 0, 1 y 2, respectivamente. Nuestra meta inicial es la siguiente: Dado un par entrada/salida, generar las direcciones de todas las salidas intermedias de ese camino, estas direcciones serán únicas. Dos caminos de entrada/salida presentan un conflicto si atraviesan el mismo enlace intermedio o comparten una salida común en una etapa intermedia.

A continuación obtendremos la condición de bloqueo para la red Omega. Para otras redes es posible establecer condiciones equivalentes de manera similar. A partir de ahora todas las numeraciones de las entradas/salidas se representarán en base k. Para facilitar el análisis supondremos una red con conmutación de circuitos. Consideremos que se ha establecido un circuito desde $s_{n-1}s_{n-2}\cdots s_1 s_0$ a $d_{n-1}d_{n-2}\cdots d_1 d_0$. Consideremos la primera etapa de enlaces en la figura 5.66. Esta parte de la red establece la siguiente conexión entre sus entradas y salidas.

$$s_{n-1}s_{n-2}\ldots s_1 s_0 \rightarrow s_{n-2}s_{n-3}\ldots s_1 s_0 s_{n-1} \tag{5.2}$$

La parte derecha de la ecuación anterior es la codificación de la salida en el patrón k-desplazamiento y la codificación de la entrada al primer nivel de conmutadores. Cada conmutador sólo puede cambiar el dígito menos significativo de la codificación actual. Después de la permutación desplazamiento, éste es s_{n-1}. Así que la salida de la primera etapa será aquella que haga que el dígito menos significativo de la codificación sea igual a d_{n-1}. Por lo tanto, la conexión entrada/salida establecida en la primera etapa de conmutadores deberá ser aquella que conecta la entrada a la salida como sigue:

$$s_{n-2}s_{n-3}\ldots s_1 s_0 s_{n-1} \rightarrow s_{n-2}s_{n-3}\ldots s_1 s_0 d_{n-1} \tag{5.3}$$

La parte derecha de esta expresión es la codificación de la entrada en la segunda

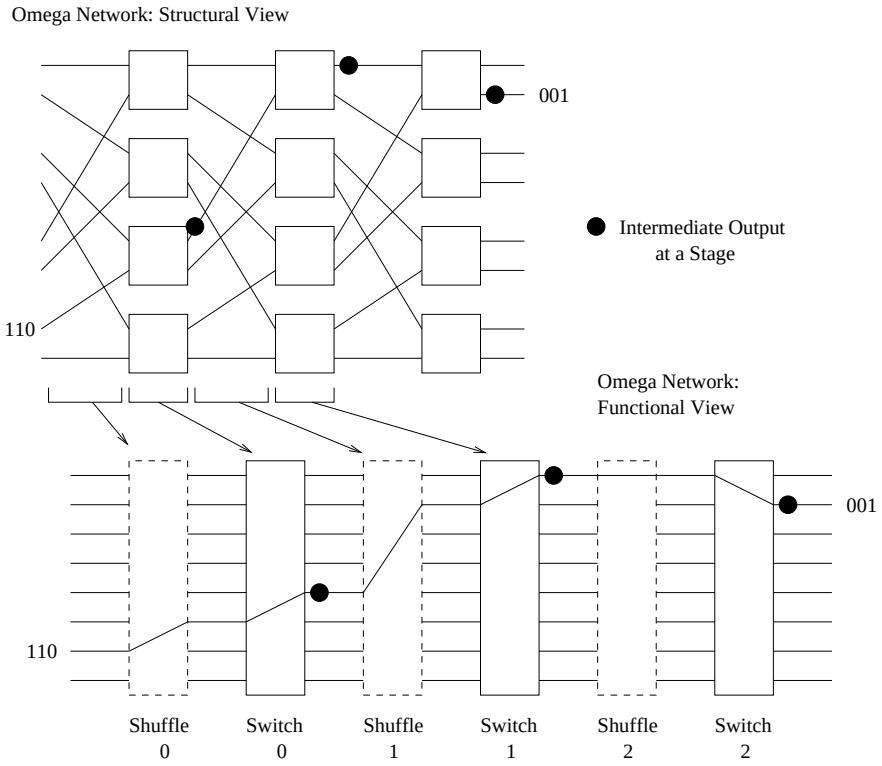


Figura 5.66: Vista funcional de la estructura de una red Omega multietapa.

etapa de enlaces. Es también la salida de la primera etapa de conmutadores y por tanto la primera salida intermedia. En la figura 5.66 se muestra un camino desde la entrada 6 a la salida 1 que tiene un total de tres salidas intermedias. De forma similar, podemos escribir la expresión de la salida intermedia en la etapa i como

$$s_{n-i-2}s_{n-i-3}\ldots s_0 d_{n-1}d_{n-2}\ldots d_{n-i-1} \tag{5.4}$$

Esto es suponiendo que las etapas están numeradas desde 0 hasta $n-1$. Ahora podemos escribir la condición de bloqueo. Para cualquier entrada/salida (S, D) y (R, T), los dos caminos pueden establecerse sin conflictos si y sólo si, $\forall i,\ 0 \leq i \leq n-1$

$$s_{n-i-2}s_{n-i-3}\ldots s_0 d_{n-1}d_{n-2}\ldots d_{n-i-1} \neq r_{n-i-2}r_{n-i-3}\ldots r_0 t_{n-1}t_{n-2}\ldots t_{n-i-1} \tag{5.5}$$

La detección de bloqueos en dos pares de entrada/salida no es tan complicada como podría parecer a primera vista. Observando la estructura de la condición de bloqueo podemos observar que si los circuitos se bloquean, entonces los dígitos $n-i-1$ menos significativos de las direcciones origen son iguales y los $i+1$ dígitos más significativos de las direcciones destino son iguales. Supongamos que tenemos una función $\phi(S, R)$ que devuelve el mayor entero l tal que los l dígitos menos significativos de S y R son iguales. De forma similar, supongamos que tenemos una función $\psi(D, T)$ que devuelve el mayor entero m tal que los m dígitos más significativos de D y T son iguales. Entonces dos caminos (S, D) y (R, T) pueden establecerse sin conflictos si y sólo si

$$\phi(S, R) + \psi(D, T) < n \tag{5.6}$$

donde $N = k^n$. Desde un punto de vista práctico, el bloqueo se puede calcular mediante secuencias de operaciones de desplazamiento y comparación, como se indica a continuación:

$$\begin{array}{l} s_{n-1}s_{n-2}\ldots\boxed{s_2s_1s_0d_{n-1}d_{n-2}\ldots d_3}\,d_2d_1d_0 \\ r_{n-1}r_{n-2}\ldots\boxed{r_2r_1r_0\ t_{n-1}t_{n-2}\ldots t_3}\,t_2t_1t_0 \end{array}$$

Las direcciones de los dos pares entrada/salida se concatenan. Figurativamente, una ventana de tamaño n se desplaza sobre ambos pares y se compara el contenido de ambas ventanas. Si son iguales en algún punto, entonces existe un conflicto en alguna etapa. El orden de ejecución es de $O(log_kN)$. Para determinar si todos los caminos pueden establecerse sin conflictos, deberíamos realizar $O(N^2)$ comparaciones cada una de las cuales tarda $O(log_kN)$ pasos en realizarse dando lugar a un algoritmo de orden $O(N^2log_kN)$. En comparación, el mejor algoritmo de configuración de todos los conmutadores mediante un control centralizado necesita un tiempo de $O(Nlog_kN)$. Sin embargo, cuando se utiliza la formulación arriba indicada, muy a menudo no es necesario tener en cuenta el número de comparaciones en el peor de los casos. A menudo, la propia estructura del patrón de comunicaciones puede utilizarse para determinar si existe conflicto entre ellos.

Algoritmos de autoencaminamiento para MINs.

Una propiedad única de las redes Delta es la propiedad de autoencaminamiento (*selft-routing*). Esta propiedad permite que en estas MINs las decisiones de encaminamiento se puedan realizar en función de la dirección destino, sin importar la dirección origen. El autoencaminamiento se realiza usando etiquetas de encaminamiento. Para un conmutador $k \times k$, existen k puertos de salida. Si el valor de la etiqueta de encaminamiento correspondiente es i $(0 \leq i \leq k-1)$, el paquete correspondiente será enviado a través del puerto i. Para una MIN de n estados, la etiqueta de encaminamiento es $T = t_{n-1}\ldots t_1t_0$, donde t_i controla el conmutador de la etapa G_i.

Cada conmutador sólo es capaz de cambiar el dígito menos significativo de la dirección actual, Por lo tanto, las etiquetas de encaminamiento tendrán en cuenta qué dígito es el menos significativo en cada etapa, reemplazándolo por el correspondiente dígito de la dirección destino. Para un destino dado $d_{n-1}d_{n-2}\ldots d_0$, la etiqueta de encaminamiento en una red mariposa se forma haciendo $t_i = d_{i+1}$ para $0 \leq i \leq n-2$ y $t_{n-1} = d_0$. En una MIN cubo, la etiqueta está formada por $t_i = d_{n-i-1}$ para $0 \leq i \leq n-1$. Finalmente, en una red Omega, la etiqueta de encaminamiento se forma haciendo $t_i = d_{n-i-1}$ para $0 \leq i \leq n-1$. El siguiente ejemplo muestra los caminos seleccionados por el algoritmo de encaminamiento basado en etiquetas para una MIN mariposa de 16 nodos.

Ejemplo

La figura 5.67 muestra una red MIN de 16 nodos usando conmutadores 2×2, y los caminos seguidos por los paquetes del nodo 0100 al nodo 0011 y del nodo 1010 al nodo 1000. Como se indicó anteriormente, la etiqueta de encaminamiento para un destino dado $d_{n-1}d_{n-2}\ldots d_0$ se forma haciendo $t_i = d_{i+1}$ para $0 \leq i \leq n-2$ y $t_{n-1} = d_0$. Así, la etiqueta para el destino 0011 es 1001. Esta etiqueta indica que el paquete debe elegir la salida superior del

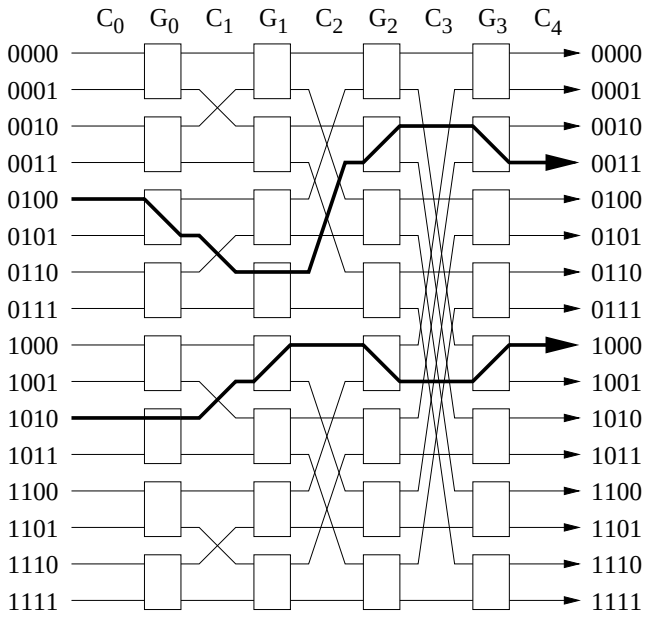


Figura 5.67: Selección del camino por el algoritmo de encaminamiento basado en etiquetas en una MIN mariposa de 16 nodos.

conmutador (puerto 0) en los estados G_2 y G_1, y la salida inferior (puerto 1) en los estados G_3 y G_0. La etiqueta para el destino 1000 es 0100. Esta etiqueta indica que el paquete debe coger la salida superior de conmutador en los estados G_3, G_1y G_0, y la salida inferior del conmutador en el estado G_2.

Una de las características interesantes de las MINs tradicionales vistas anteriormente (TMINs) es que existe un algoritmo sencillo para encontrar un camino de longitud $log_k N$ entre cualquier par entrada/salida. Sin embargo, si un enlace se congestiona o falla, la propiedad de camino único puede fácilmente interrumpir la comunicación entre algunos pares entrada/salida. La congestión de paquetes sobre algunos canales causa el problema conocido como punto caliente. Se han propuesto muchas soluciones para resolver este problema. Una aproximación popular es proporcionar múltiples caminos alternativos entre cualquier par origen y destino con lo que se consigue reducir la congestión de la red al mismo tiempo que se consigue tolerancia a fallos. Estos métodos requieren normalmente hardware adicional, como el uso de etapas extras o canales adicionales.

El uso de canales adiciones da lugar a las MINs dilatadas. En una MIN d-dilatada (DMIN), cada conmutador se reemplaza por un conmutador d-dilatado. En este conmutador, cada puerto tiene d canales. Usando canales replicados, las DMINs ofrecen una mejora sustancial del *throughput* de la red. La etiqueta de encaminamiento en una DMIN puede determinarse mediante la dirección destino al igual que se mencionó para las TMINs. Dentro de cada conmutador, los paquetes se envían a un puerto de salida particular de forma aleatoria entre los canales libres. Si todos los canales están ocupados, el paquete se bloquea.

Otra aproximación al diseño de MINs consiste en permitir comunicaciones bidireccionales. En este caso, cada puerto del conmutador tiene canales duales. En una red mariposa (BMIN) construida con conmutadores $k \times k$, la dirección origen S y la dirección destino D se representan mediante números k-arios $s_{n-1} \ldots s_1 s_0$ y $d_{n-1} \ldots d_1 d_0$

respectivamente. La función $FirstDifference\,(S, D)$ devuelve t, la posición en donde existe la primera diferencia (más a la izquierda) entre $s_{n-1} \ldots s_1 s_0$ y $d_{n-1} \ldots d_1 d_0$.

Un camino de encaminamiento con vuelta atrás entre cualquier origen y destino debe cumplir las siguientes restricciones:

- El camino consiste en algunos canales hacia delante, algunos canales hacia atrás, y exactamente una conexión de vuelta.
- El número de canales hacia adelante es igual al número de canales hacia atrás.
- Ningún canal hacia adelante y hacia atrás a lo largo del camino son el canal pareja del mismo puerto.

Obsérvese que la última condición se utiliza para prevenir comunicaciones redundantes. Para encaminar un paquete desde el origen al destino, el paquete se envía en primer lugar a la etapa G_t. No importa a que conmutador llegue (en la etapa G_t). Después, el paquete gira y se envía de vuelta a su destino. Cuando se mueve hacia el estado G_t, un paquete tiene varias opciones sobre qué canal de salida elegir. La decisión puede resolverse seleccionando aleatoriamente entre aquellos canales de salida que no están bloqueados por otros paquetes. Después de que el paquete ha llegado a un conmutador en el estado G_t, elige el único camino desde ese conmutador hacia su destino. Este camino de vuelta puede determinarse mediante un algoritmo de encaminamiento basado en etiquetas para TMINs.

Observar que los algoritmos de encaminamiento descritos arriba son algoritmos distribuidos, en donde cada conmutador determina el canal de salida basándose en la información que lleva el propio paquete. El siguiente ejemplo muestra los caminos disponibles en una red BMIN mariposa de ocho nodos.

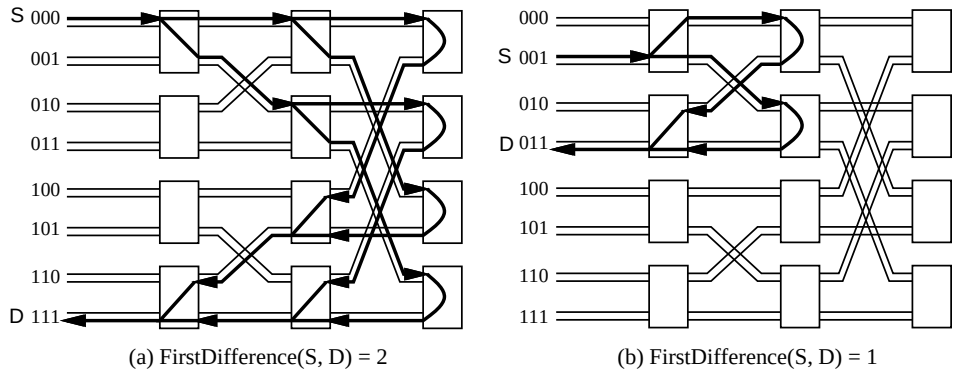


Figura 5.68: Caminos disponibles en una BMIN mariposa de ocho nodos.

Ejemplo

La figura 5.68 muestra los caminos disponibles para una red BMIN mariposa de ocho nodos usando conmutadores 2×2. La figura 5.68a muestra el caso en donde los nodos origen y destino son 000 y 111, respectivamente. En este caso $FirstDifference(S, D) = 2$. Así, los paquetes giran en la etapa G_2 y existen cuatro caminos diferentes disponibles. En la figura 5.68b, los nodos origen y destino son 001y 011, respectivamente. $FirstDifference(S, D) = 1$, y el paquete gira en la etapa G_1.

5.7.8. Resumen de las redes indirectas y equivalencias

En la tabla 5.2 se resumen las características más importantes de los buses, redes multietapa, y conmutadores de barra cruzada, para la construcción de redes dinámicas. Evidentemente el bus es la opción más económica pero tiene el problema del bajo ancho de banda disponible para cada procesador, especialmente si hay muchos. Otro problema con el bus es que es sensible a los fallos, de manera que es necesario su duplicación para obtener sistemas tolerantes a fallos.

Características de la red	Bus	Multietapa	Barra cruzada
Latencia mínima por unidad de transmisión	Constante	$O(\log_k n)$	Constante
Ancho de banda por procesador	de $O(w/n)$ a $O(w)$	de $O(w)$ a $O(nw)$	de $O(w)$ a $O(nw)$
Complejidad cableado	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Complejidad conmutadores	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Capacidad de rutado y conectado	Sólo uno a uno cada vez.	Algunas permutaciones y broadcast si la red no está bloqueada.	Todas las permutaciones a la vez.
Computadores representativos	Symmetry S-1 Encore Multimax	BBN TC-2000 IBM RP3	Cray Y-MP/816 Fujitsu VPP500
Notas	Suponemos n procesadores; la anchura de banda es w	MIN $n \times n$ con conmutadores $k \times k$ y líneas de w bits	Barra cruzada $n \times n$ con líneas de w bits

Cuadro 5.2: Resumen de las características de las redes dinámicas.

El conmutador de barra cruzada es el más caro de hacer, especialmente teniendo en cuenta que su complejidad hardware crece con n^2. Sin embargo, la barra cruzada tiene el mayor ancho de banda y las mayor capacidad de rutado. Para una red con no demasiados elementos es la mejor opción.

Las redes multietapa suponen un compromiso intermedio entre los dos extremos. La mayor ventaja de las MINs es su escalabilidad gracias a la construcción modular. El problema de esta red es que la latencia crece con $\log n$, es decir, con el número de etapas en la red. También, el coste asociado con los cables y los conmutadores puede ser importante.

Para la construcción de futuros MPP, algunas de las topologías estáticas pueden resultar más interesantes por su alta escalabilidad en determinadas aplicaciones. Si las tecnologías ópticas y microelectrónicas avanzan rápidamente, los MINs a escala alta o las redes de barras cruzadas podrían llegar a ser realizables o económicamente rentables para la realización de conexiones dinámicas en computadores de propósito general.

5.8. Conclusiones

Los procesadores simétricos de memoria compartida son una extensión natural de las estaciones de trabajo uniprocesador y de los PCs. Las aplicaciones secuenciales pueden ejecutarse sin necesidad de modificación, y beneficiándose incluso en el rendimiento al tener el procesador por una fracción de tiempo mayor y de una mayor cantidad de memoria principal compartida y capacidades de E/S típicamente disponibles en estas máquinas. Las aplicaciones paralelas son también fácilmente portables a este entorno, ya que todos los datos locales y compartidos son directamente accesibles por todos los procesadores utilizando operaciones de lectura y escritura ordinarias. Las porciones de cálculo intensivo de una aplicación secuencial pueden ser paralelizadas de forma selectiva. Una ventaja clave de las cargas de multiprogramación es la granularidad fina con la que se pueden compartir los recursos entre los distintos procesos. Esto es cierto tanto en lo temporal, donde procesadores y/o páginas de la memoria principal pueden ser reasignadas frecuentemente entre los diferentes procesos, como en lo físico, donde la memoria principal puede dividirse entre las aplicaciones a nivel de página. Debido a estas características, los principales vendedores de ordenadores, desde las estaciones de trabajo suministradas por SUN, Silicon Graphics, Hewlett Packard, Digital e IBM hasta los suministradores de PCs como Intel y Compaq están produciendo y vendiendo este tipo de máquinas. De hecho, para algunos de los vendedores de estaciones de trabajo este mercado constituye una fracción substancial de sus ventas, y una mayor fracción de sus beneficios netos debido a los mayores márgenes existentes en estas máquinas.

El reto clave en el diseño de estas máquinas es la organización y la implementación del subsistema de memoria compartida usado como comunicación entre los procesos además de manejar el resto de accesos a memoria. La mayoría de las máquinas de pequeña escala que se encuentran en la actualidad usan un bus como interconexión. En este caso el reto está en mantener coherentes en las cachés privadas de los procesadores los datos compartidos. Existen una gran cantidad de opciones disponibles, como hemos discutido, incluyendo el conjunto de estados asociados a los bloques de la caché, la elección del tamaño del bloque, y si usar invalidación o actualización. La tarea clave de la arquitectura del sistema es realizar las elecciones oportunas en función de los patrones de compartición de datos predominantes en las aplicaciones para las que se utilizará la máquina y de su facilidad de implementación.

Conforme progresan los procesadores, el sistema de memoria, los circuitos integrados y la tecnología de empaquetamiento, existen tres importantes razones que nos hace pensar que los multiprocesadores de pequeña escala continuarán siendo importantes en un futuro. La primera es que ofrecen un buen coste/rendimiento que hará que individuos o pequeños grupos de personas pueden utilizarlos como un recurso compartido o un servidor de cómputo o ficheros. La segunda es que los microprocesadores actuales están diseñados para poder ser utilizados como multiprocesadores por lo que ya no existe un tiempo de espera significativo entre el último microprocesador y su incorporación a un multiprocesador. La tercera razón es que la tecnología software para máquinas paralelas (compiladores, sistemas operativos, lenguajes de programación) está madurando rápidamente para máquinas de memoria compartida de pequeña escala, aunque todavía le queda un largo camino en el caso de máquinas de gran escala. Por ejemplo, la mayoría de los vendedores de sistemas de ordenadores tienen una versión paralela de sus sistemas operativos para sus multiprocesadores basados en bus.