Capítulo 6

Multiprocesadores de Memoria Compartida

6.1 Introducción

La arquitectura paralela prevalente es el multiprocesador de pequeña o moderada escala que proporciona un espacio de direccionamiento global y acceso uniforme a toda la memoria principal desde cualquier procesador. A esta arquitectura se le denomina Multiprocesador Simétrico o SMP. Cada procesador tiene su propia cache y todos los procesadores y módulos de memoria están conectados a la misma red de interconexión, usualmente un bus compartido. Los SMPs dominan el mercado de los servidores y son cada vez más comunes. También son bloques de construcción importantes para sistemas con más procesadores. La compartición eficiente de los recursos, como la memoria y los procesadores, así como la posibilidad de acceder a todos los datos compartidos de forma eficiente desde cualquier procesador usando operaciones de lectura y almacenamiento hacen a este tipo de máquinas atractivas tanto para entornos multitarea como para la programación paralela. El movimiento y replicación automática de los datos compartidos en las cachés locales permiten facilitar la tarea del programador. Estas características también son muy útiles en el sistema operativo, cuyos diferentes procesos comparten estructuras de datos y pueden ejecutarse en diferentes procesadores.

Dado que todas las comunicaciones y cálculos locales dan lugar a accesos a memoria en un espacio de direcciones compartido, desde la perspectiva de la arquitectura del sistema el elemento de diseño clave es la organización de una jerarquía de memoria extendida. En general, las jerarquías de memoria en los multiprocesadores caen dentro de una de las cuatro categorías que se muestran en la figura 6.1. Estas categorías se corresponden aproximadamente con el tamaño del multiprocesador que estemos considerando. Las tres primeras se corresponden con procesadores simétricos, lo que no ocurre con la cuarta categoría.

En la primera categoría, la aproximación de caché compartida (figura 6.1a), la red de interconexión se localiza entre los procesadores y un primer nivel de caché compartida, que a su vez está conectado al subsistema de memoria principal. Tanto la caché como el sistema de memoria principal pueden estar entrelazados para incrementar el ancho de banda disponible. Esta aproximación se ha venido usando para conectar un número muy pequeño de procesadores (2-8). A mediados de los 80 fue una técnica común para conectar una pareja de procesadores a una placa; hoy en día, es una posible estrategia para encapsular un multiprocesador en un único chip, donde un número pequeño de procesadores

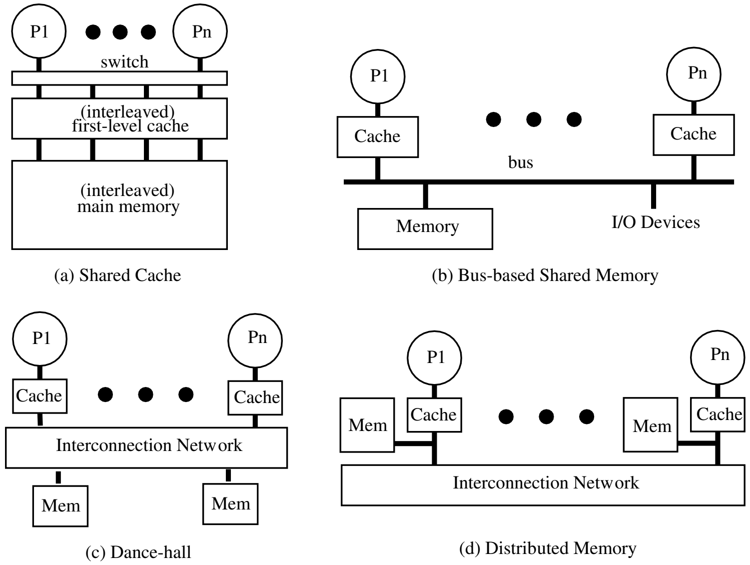


Figura 6.1: Jerarquías de memoria habituales en multiprocesadores.

en el mismo chip comparte el primer nivel de caché. Sin embargo, se aplica a muy pequeña escala debido a que la red de interconexión entre los procesadores y el primer nivel de caché se encuentra en el camino crítico que determina la latencia de acceso a caché y a que la caché compartida debe de proporcionar un ancho de banda tremendo para que los procesadores la aprovechen.

En la segunda categoría, la aproximación de memoria compartida mediante bus (figura 6.1b), la red de interconexión es un bus compartido localizado entre las cachés privadas de cada uno de los procesadores y el subsistema de memoria principal. Esta aproximación ha sido ampliamente usada para multiprocesadores de pequeña y mediana escala formados por hasta 20 ó 30 procesadores. Es la forma dominante de máquina paralela vendida hoy en día, y en la actualidad todos los modernos microprocesadores han invertido un considerable esfuerzo de diseño para soportar configuraciones de memoria compartida con "coherencia de caché". Por ejemplo, el procesador Intel Pentium Pro puede conectarse a un bus compartido sin ninguna lógica adicional. El principal factor que limita la escalabilidad en estas máquinas es la limitación en el ancho de banda del bus compartido y del sistema de memoria.

Las últimas dos categorías están pensadas para soportar un gran número de nodos. La tercera categoría, la aproximación de *salón de baile* (dance-hall), también inserta una red de interconexión entre las cachés y la memoria principal, pero la red es ahora una red punto-a-punto escalable y la memoria se divide en módulos lógicos que se conectan a diferentes puntos de la red. Esta aproximación es simétrica, toda la memoria principal está a igual distancia de todos los procesadores, pero su limitación es que toda la memoria está igual de *lejos* de todos los procesadores: En sistemas grandes es necesario realizar varios saltos o atravesar varios conmutadores en la red de interconexión para alcanzar cualquier módulo de memoria. La cuarta categoría, la aproximación de "memoria distribuida", no es simétrica: Existe una red de interconexión escalable entre los nodos de procesamiento pero cada nodo tiene su porción local de la memoria principal global a la cual tiene un acceso más rápido (Figura 6.1d). Explotando la localidad en la distribución de datos, esperamos que la mayoría de accesos se satisfagan en la memoria local y no tengan que atravesar la red. Este diseño es, por lo tanto, más atractivo para multiprocesadores escalables. Por supuesto, también es posible combinar varias aproximaciones en un mismo diseño - por ejemplo, una máquina de memoria distribuida cuyos nodos individuales son SMP de bus compartido.

En todos los casos, las cachés juegan un papel esencial en la reducción del tiempo de acceso a la memoria visto por el procesador y en reducir el ancho de banda que cada procesador utiliza de la red de interconexión y del sistema de memoria. La necesidad de ancho de banda se reduce debido a que los accesos a datos emitidos por un procesador son satisfechos en la caché y no deben de aparecer en el bus; en caso contrario, cada acceso de cualquier procesador aparecería en el bus que se saturaría rápidamente. En todas las aproximaciones menos en la de caché compartida, cada procesador tiene al menos un nivel de su jerarquía de caché que es privada y que da lugar a un problema denominado *coherencia de caché*. El problema surge cuando un bloque de memoria está presente en la caché de uno o más procesadores y otro procesador modifica ese bloque de memoria. Al menos que se realicen acciones especiales, los procesadores continuarán accediendo a la copia del bloque existente en sus cachés y cuyo contenido no es correcto.

6.2 Redes de interconexión en multiprocesadores de memoria compartida

La característica principal de los sistemas multiprocesadores es la capacidad de cada procesador para compartir un conjunto de módulos de memoria y, quizá, unos dispositivos de E/S. Esta capacidad de compartición se deriva del uso de una red de interconexión. En el caso de los multiprocesadores de memoria compartida las alternativas de red de interconexión elegidas han sido los buses comunes y las redes indirectas.

6.2.1 Caracterización de las redes indirectas

Las redes indirectas se pueden modelizar 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 tema anterior, 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 como 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* (barras cruzadas). Sin embargo, el número de conexiones físicas de un conmutador está limitado por factores hardware tales como el número de pins disponibles y la densidad máxima del cableado. Estas dificultades imposibilitan el uso de crossbar en redes de gran tamaño. Como consecuencia, se han propuesto un gran número de topologías alternativas.

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

6.2.2 Red de Barras Cruzadas

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

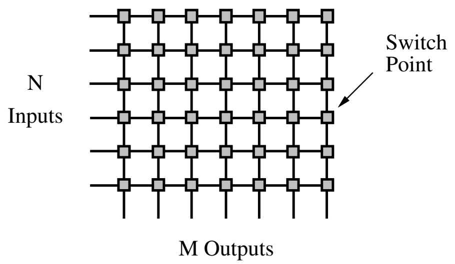


Figura 6.2: Un crossbar de $N \times M$.

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 red conmutada con N entradas y M salidas, que permite hasta $\min\{N, M\}$ interconexiones punto a punto sin contención. La figura 6.2 muestra una red crossbar de $N \times M$. Normalmente, $M = N$ excepto para 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. Las crossbars han sido utilizadas tradicionalmente en mutlprocesadores 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 arbitro 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 un 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 6.3. En la figura 6.3a, 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 6.3b, 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 6.3c, una entrada de una fila superior tiene acceso a la salida. Sin embargo, la entrada de la fila en la que se encuentra en punto de conmutación también ha solicitado esa salida y está bloqueada. La configuración de la figura 6.3d sólo es necesaria si el crossbar tiene soporte para comunicaciones *multicast* (uno a muchos).

Los avances en la tecnología 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 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 $n \times n$.

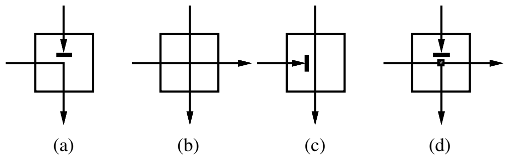


Figura 6.3: Estados de un punto de conmutación en una red de barras cruzadas.

6.2.3 Redes Multietapa

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 barras cruzadas. 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 los 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 maquinas comerciales.

6.2.4 Un modelo MIN generalizado

Existen muchas maneras diferentes de interconectar etapas adyacentes. La figura 6.4 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 , 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

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

La conexión entre dos etapas adyacentes G_{i-1} y G_i, denotada por C_i, define el *patrón de conexión*

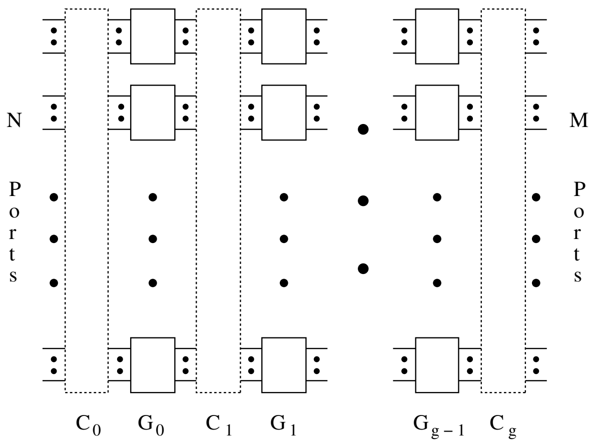


Figura 6.4: Un MIN general con N entradas, M salidas, y g etapas.

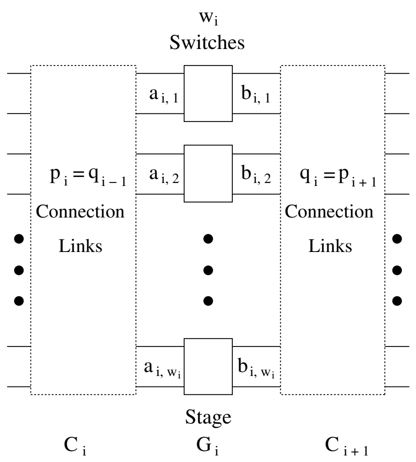


Figura 6.5: Visión detallada de una etapa G_i.

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 como 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 $p_i - 1$. En la figura 6.5 se muestra una visión detallada de la etapa G_i

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 conmutadores $k \times k$ en n etapas, donde cada etapa contiene $\frac{N}{k}$ conmutadores. Muchas de las redes multietapa más conocidas, como la Omega, flip, cube, butterfly, 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 $0 \leq x_i \leq k-1$, $0 \leq i \leq n-1$.

Conexión Baraje Perfecto

La conexión 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

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

La conexión k-*baraje perfecto* (perfect shuffle) realiza un desplazamiento cíclico hacia la izquierda de los dígitos de X en una posición. Para $k = 2$, esta acción se corresponde con el barajamiento perfecto de una baraja de N cartas, como se demuestra en la figura 6.6a para el caso de $N = 8$. El baraje perfecto corta la baraja en dos parte y las entremezcla empezando con la segunda parte. El *baraje perfecto inverso* (inverse perfect shuffle) realiza la acción contraria como se define a continuación:

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

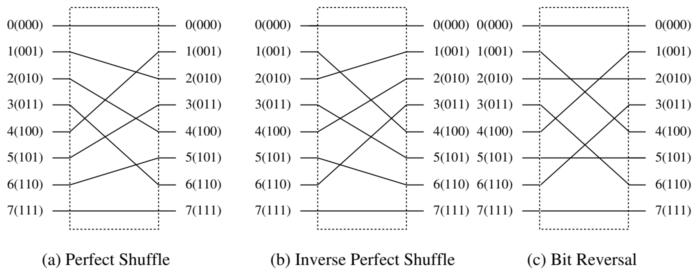


Figura 6.6: Barajadura perfecta, barajadura perfecta inversa, y bit reversal para $N = 8$.

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 6.6c 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 6.7 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.

Conexión Cubo

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

$$E_i(x_{n-1}\dots x_{i+1}x_ix_{i-1}\dots x_0) = x_{n-1}\dots x_{i+1}\overline{x}_ix_{i-1}\dots x_0$$

La i-ésima conexión cubo complementa el i-ésimo bit del índice. La figura 6.8 muestra la conexión cubo para $i = 0, 1, 2$ y $N = 8$. A E_0 también se le denomina conexión *intercambio*.

Conexión en Línea Base

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

$$\delta_i^k(x_{n-1}\dots x_{i+1}x_ix_{i-1}\dots x_1x_0) = x_{n-1}\dots x_{i+1}x_0x_ix_{i-1}\dots x_1$$

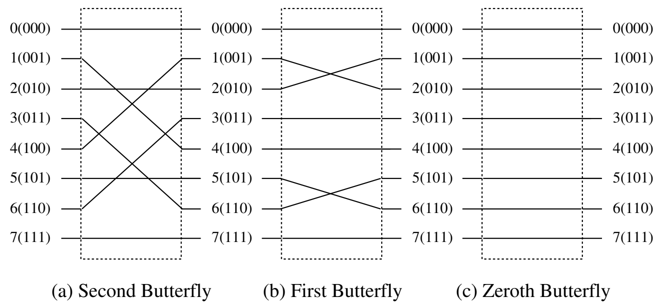


Figura 6.7: La conexión mariposa para $N = 8$.

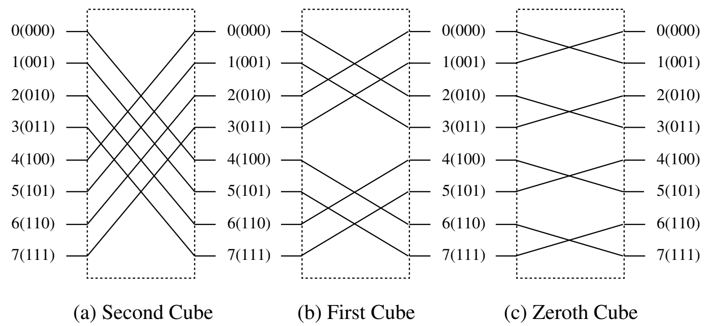
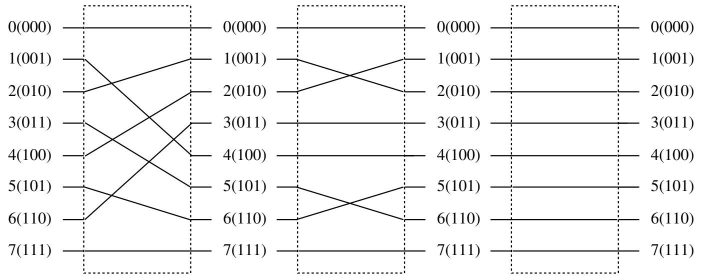


Figura 6.8: La conexión cubo para $N = 8$.

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 6.9 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.



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

Figura 6.9: La conexión en línea base para $N=8$.

6.2.5 Clasificación de las Redes de Interconexión Multietapa

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. Estas 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 número 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 de Clos, (figura 6.10) 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 6.11 muestra una red 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 las redes bloqueantes.

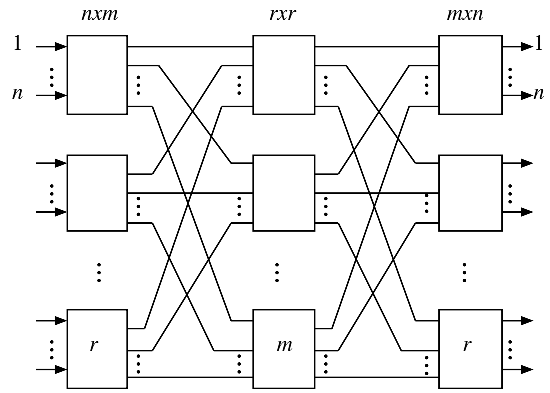


Figura 6.10: Red de Clos.

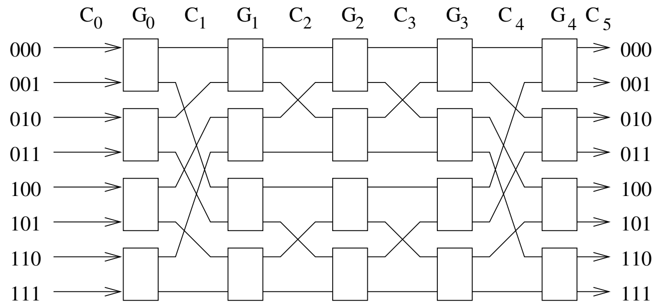


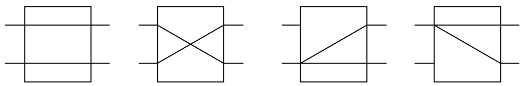
Figura 6.11: Una red Beneš 8×8.

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.

6.2.5.1 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 $\min\{a, b\}$ 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 a varios). En el modo de comunicación *broadcast* (uno a todos), cada puerto puede conectarse a todos los puertos de salida. La figura 6.12 muestra los cuatro posibles estados de un conmutados 2×2. Los últimos dos estados se usan para soportar las comunicaciones uno a muchos y uno a todos.



(a) Straight (b) Exchange (c) Lower Broadcast (d) Upper Broadcast

Figura 6.12: 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 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 todas 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.

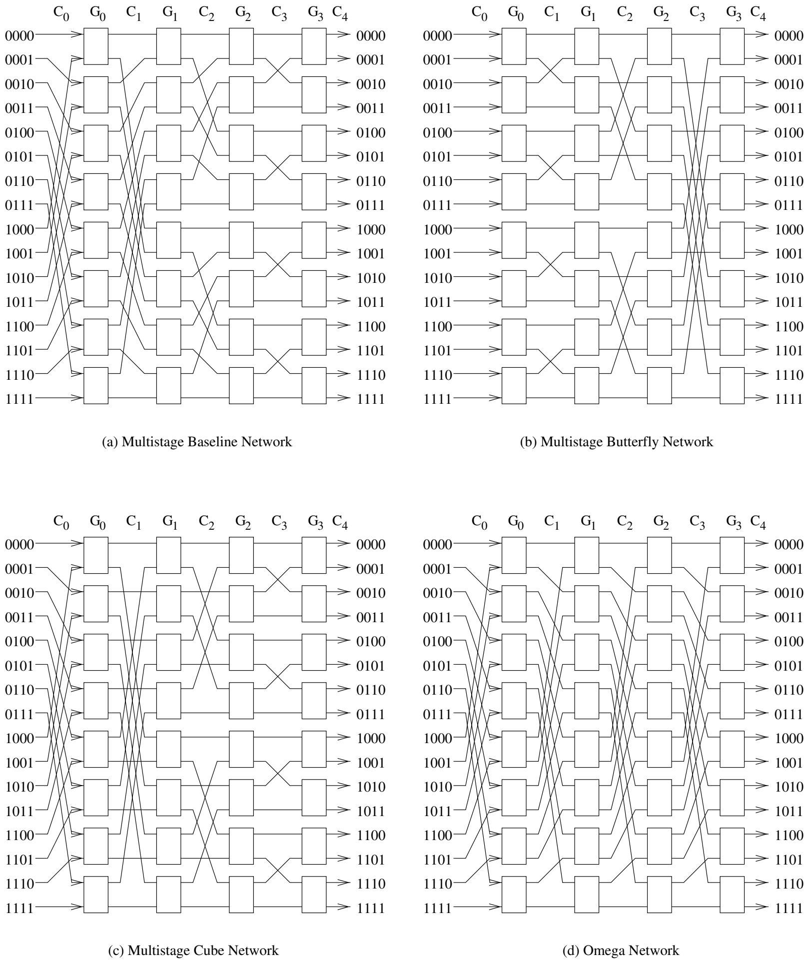
Redes multietapa de línea base (*Baseline MINs*). En una red multietapa de línea base, el patrón de conexión C_i viene descrito por la $(n - i)$-ésima permutación en línea base δ^k_{n-i} para $1 \leq i \leq n$. El patrón de conexión para C_o viene dado por σ^k.

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^k_i para $0 \leq i \leq n - 1$. El patrón de conexión de C_n es β^k_0.

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^k_{n-i} 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 β^k_0. Así, todos los patrones de conexión menos el último son idénticos. El último patrón de conexión no produce una permutación.

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 $s_{n-1} \ldots s_0$,



(a) Multistage Baseline Network (b) Multistage Butterfly Network

(c) Multistage Cube Network (d) Omega Network

Figura 6.13: Cuatro redes de interconexión unidireccionales multietapa de 16×16.

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

6.2.5.2 Redes de interconexión multietapa bidireccionales

La figura 6.14 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 6.15. Un conmutador bidireccional soporta tres tipos de conexiones: *hacia delante*, *hacia atrás* y *de vuelta* (ver figura 6.14). Dado que son posibles la conexiones de vuelta entre puertos del mismo lado del conmutador, los caminos tienen diferentes longitudes. En la figura 6.15 se muestra una MIN mariposa bidireccional de ocho nodos.

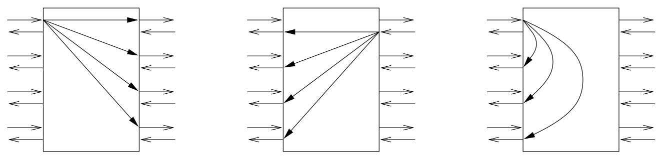


Figura 6.14: Conexiones en un conmutador bidireccional.

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 6.16 muestra dos caminos alternativos desde el nodo S al nodo D en una BMIN mariposa de ocho nodos. Cuando se cruzan las etapas hacia adelante, 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 6.17, 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

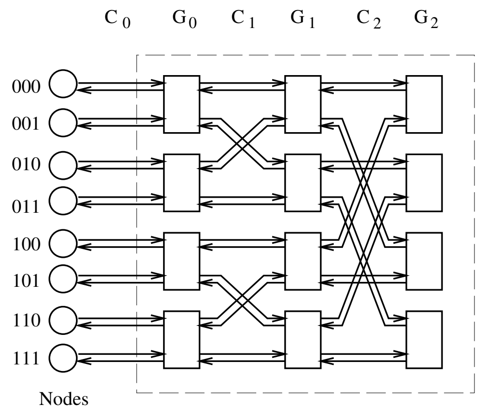


Figura 6.15: Una MIN mariposa bidireccional de ocho nodos.

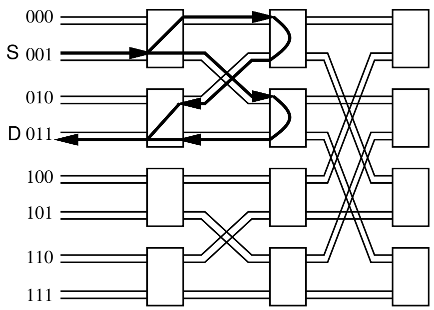
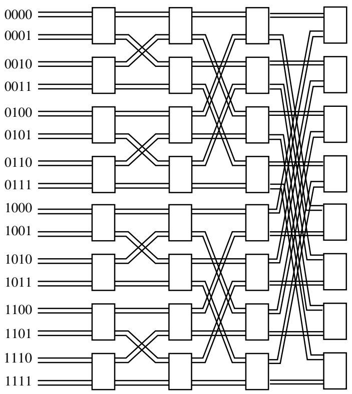
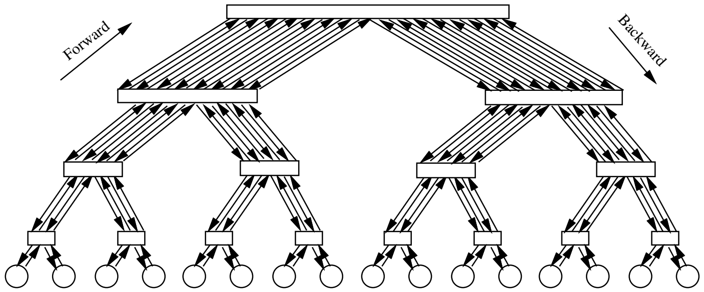


Figura 6.16: Caminos alternativos para una MIN mariposa bidireccional de ocho nodos.



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



(b) A 16−Node Fat Tree

Figura 6.17: Árbol grueso y BMIN mariposa.

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.

6.2.6 Encaminamiento en redes multietapa (MINs)

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 de 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 encaminamiento 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 $n = log_k N$etapas, 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 de 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 6.18 para el caso de una red Omega con conmutadores 2×2. Cada etapa de conmutadores y cada etapa de enlaces pueden 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

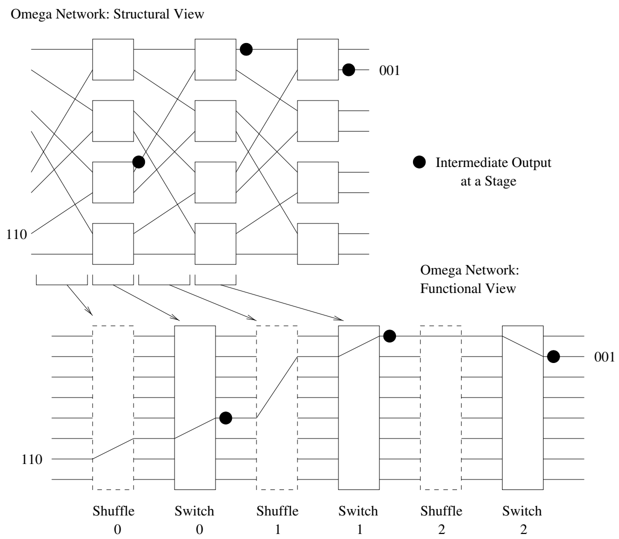


Figura 6.18: Vista funcional de la estructura de una red Omega multietapa.

$d_{n-1}d_{n-2}\cdots d_1d_0$. Consideremos la primera etapa de enlaces en la figura 6.18. Esta parte de la red establece la siguiente conexión entre sus entradas y salidas.

$$s_{n-1}s_{n-2}\dots s_1s_0 \rightarrow s_{n-2}s_{n-3}\dots s_1s_0s_{n-1} \tag{6.1}$$

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:

$$s_{n-2}s_{n-3}\dots s_1s_0s_{n-1} \rightarrow s_{n-2}s_{n-3}\dots s_1s_0d_{n-1} \tag{6.2}$$

La parte derecha de esta expresión es la codificación de la entrada en la segunda etapa de enlaces. Es también la salida de la primera etapa de conmutadores y por tanto la primera salida intermedia. En la figura 6.18 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

$$s_{n-i-2}s_{n-i-3}\dots s_0d_{n-1}d_{n-2}\dots d_{n-i-1} \tag{6.3}$$

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, $\forall i,\ 0 \leq i \leq n-1$

$$s_{n-i-2}s_{n-i-3}\dots s_0d_{n-1}d_{n-2}\dots d_{n-i-1} \neq r_{n-i-2}r_{n-i-3}\dots r_0t_{n-1}t_{n-2}\dots t_{n-i-1} \tag{6.4}$$

La detección de bloqueos en dos pares de entrada/salida no es tan complicado 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

$$\phi(S, R) + \psi(D, T) < n \tag{6.5}$$

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:

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

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 $O(log_k N)$. Para determinar si todos los caminos pueden establecerse sin conflictos, deberíamos realizar $O(N^2)$ comparaciones cada una de las cuales tarda $O(log_k N)$ pasos en realizarse dando lugar a un algoritmo de orden $O(N^2 log_k N)$. En comparación, el mejor algoritmo de configuración de todos los conmutadores mediante un control centralizado necesita un tiempo de $O(N log_k N)$. Sin embargo, cuando se utiliza la formulación arriba indicada muy a menudo no es necesario realizar las 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 $T = t_{n-1} \dots t_1 t_0$, 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 que dígito es el menos significativo en cada etapa, reemplazandolo por el correspondiente dígito de la dirección destino. Para un destino dado $d_{n-1}d_{n-2} \dots d_0$, 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.

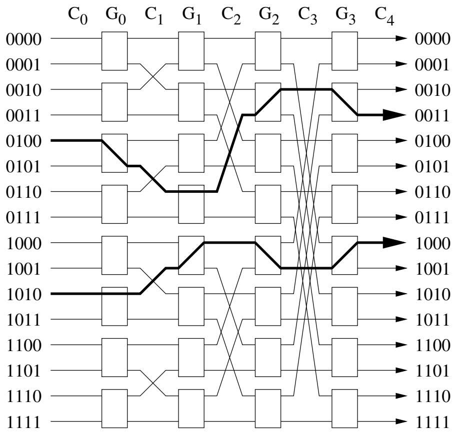


Figura 6.19: Selección del camino por el algoritmo de encaminamiento basado en etiquetas en una MIN mariposa de 16 nodos.

Ejemplo

> La figura 6.19 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 $d_{n-1}d_{n-2} \dots d_0$ 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 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, DMINs ofrecen una mejora sustancial del *throughput* (rendimeinto) 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 envía 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 $s_{n-1} \dots s_1 s_0$ y $d_{n-1} \dots d_1 d_0$ respectivamente. La función $FirstDifference(S, D)$ devuelve *t*, la posición en donde existe la primera diferencia (más a la izquierda) entre $s_{n-1} \dots s_1 s_0$ y $d_{n-1} \dots d_1 d_0$.

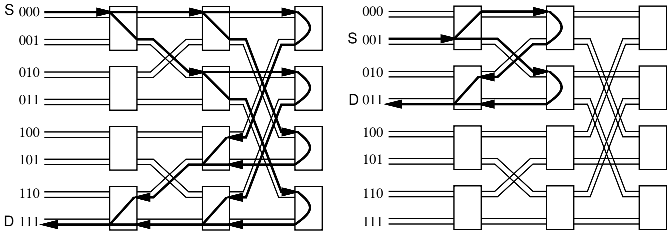
Un camino para ruteo con vuelta atrás entre cualquier origen y destino debe de cumplir las siguientes restricciones:

- 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 es 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 hasta 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 que canal de salida a 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.

Ejemplo



(a) FirstDifference(S, D) = 2 (b) FirstDifference(S, D) = 1

Figura 6.20: Caminos disponibles en una BMIN mariposa de ocho nodos.

La figura 6.20 muestra los caminos disponibles para una red BMIN mariposa de ocho nodos usando conmutadores 2×2. La figura 6.20a 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 6.20b, los nodos origen y destino son 001y 011, respectivamente. $FirstDifference(S, D) = 1$, y el paquete gira en la etapa G_1.

6.3 Coherencia de las cachés

Pensemos por un momento cual 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.

6.3.1 El problema de la coherencia de las cachés

Para ilustrar este tipo de inconsistencia, consideremos la arquitectura que se muestra en la figura 6.21. 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 cuales son los valores leídos por P_1 y P_2 dependiendo de tipo de caché utilizada.

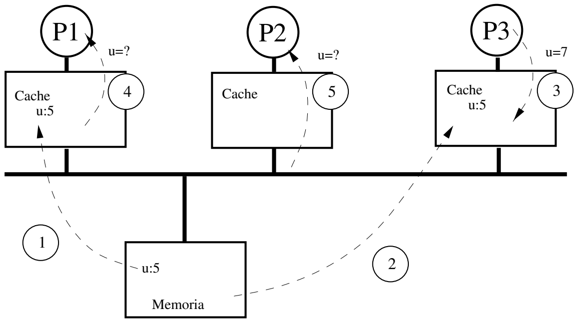


Figura 6.21: Ejemplo del problema de coherencia de las cachés.

- 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 se 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 memora 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 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.

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 de 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 ordenes de programas separados para cada proceso y esos ordenes de programa interactúan cuando acceden al sistema de memoria. Una manera de agudizar nuestra noción intuitiva de sistema de memoria coherente es determinar que 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 procesdores en dicho punto, así la memoría 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 intercalamiento que preserve la ordenación de cada programa individual es razonable. Podemos asumir algunos hechos básicos; las operaciones de cada procesador serán realizadas en algún momento dado. Además, nuestra noción intuitiva de

“último” puede ser visto como el más reciente en alguna de las hipotéticas ordenaciones secuenciales que mantienen las propiedades discutidas en este párrafo.

Dado que este orden secuencial debe de 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.

Informalmente, se dice que un sistema de memoria es coherente si cualquier lectura de una posición devuelve el valor más recientemente escrito en esa posición. Esta definición, aunque intuitiva es vaga y simple: la realidad es mucho más compleja. Esta definición contiene dos aspectos diferentes del comportamiento del sistema de memoria. El primero, denominado coherencia define qué valores pueden ser devueltos por una lectura. El segundo, llamado consistencia, determina cuándo un valor escrito será devuelto por una lectura.

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 ocurren 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 de 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 que orden las cosas ocurren realmente en la máquina o cuando cambio 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 completación de una operación: Una operación de lectura o escritura se dice que *se ha completado* cuando se ha realizado con respecto a todos los procesadores.

6.3.2 Esquemas de coherencia de las cachés

En general existen 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 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 *protocolos de caché basados en 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*.

Los protocolos de coherencia de caché basados en *snoopy* se estudiarán en este capítulo, los basados en la existencia de un directorio los dejaremos para el capítulo de Multiprocesadores Escalables.

6.4 Modelos de consistencia de la memoria

La coherencia es esencial si se quiere transmitir información entre procesadores mediante la escritura de un procesador en una localización que leerá el otro procesador. Sin embargo, 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 6.22 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 de 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 valor escrito" para cada localización: Para establecer un orden entre los accesos realizados por diferentes procesos a


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

Figura 6.22: Requisitos de un evento de sincronización mediante flags.

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.

6.4.1 Consistencia secuencial

Un modelo de ordenación deseable para un espacio de direcciones compartido es el que nos asegura que el resultado de un programa multihilo (*multithread*) ejecutándose con cualquier solapamiento de los distintos hilos sobre un uniprocesador 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 el del programa, y entre los procesos la ordenación viene dada por alguno de los posibles solapamientos entre los ordenes de los diferentes programas. Así, en el caso del multiprocesador, este no debe de ser capaz de hacer que los resultados sean visibles en el espacio de direcciones compartido en una 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 6.23 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.

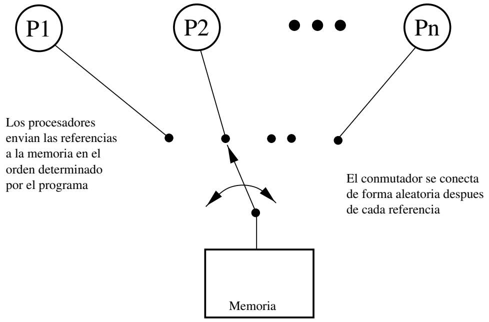


Figura 6.23: Abstracción desde el punto de vista del programador del subsistema de memoria bajo el modelo de consistencia secuencial.

Al igual que en el caso de la coherencia, no es importante en que orden se emitan realmente las operaciones a la memoria o incluso cuando se completan. Lo que importa es que parezcan completarse en un orden que no viole la consistencia secuencial. Veamoslo mediante el ejemplo que se muestra en la figura 6.24.

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

Figura 6.24: Ordenes entre los accesos sin sincronización.

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 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) preserven las restricciones intuitivas siguientes. 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 haya 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 de 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 de 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 6.25. Mostrar como 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 6.25: 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 ordenes 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 ordenes totales (intercalados como se ha definido anteriormente). Es decir, debe existir un orden total o intercalación de ordenes de los programas de los procesos que de 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.

6.4.2 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 a 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 de 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 la 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.

6.4.3 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 6.26 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 interface 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].

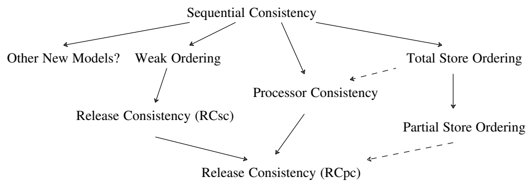


Figura 6.26: Modelos de consistencia de memoria.

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. $R \to R$: una lectura seguida de una lectura.
2. $R \to W$: 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. $W \to W$: 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. $W \to R$: 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 ordenes deben preservarse. Un modelo de consistencia secuencial exige que se preserven los cuatro ordenes 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 $W \to R$

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

significa que permitimos que una lectura que era posterior a una escritura se complete antes. Hay que recordar que una 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 que 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 al 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 de 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 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 $S \rightarrow S$.

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 haya 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 6.27 resumiremos todos los modelos, mostrando los ordenes impuestos, y mostraremos un ejemplo en la figura 6.28.

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.

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* (ordenación débil), 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 6.27, 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 necesitan 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 6.27: Ordenaciones impuestas por varios modelos de consistencia. Se muestran tanto los accesos ordinarios como los accesos a variables de sincronización.

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 de 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 6.27; en el ejemplo de la figura 6.28, se muestra como este modelo impone el menor número de ordenes de los cinco modelos.

Para comparar los modelos de *release consistency* y *weak ordering*, consideremos qué ordenaciones son necesarias para este último modelo, para ello descompondremos cada S en S_A y S_R. Esto nos

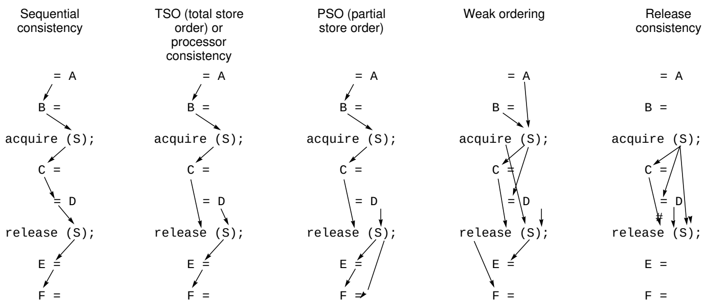


Figura 6.28: Ejemplos de los cinco modelos de consistencia vistos en esta sección que muestra la reducción en el número de ordenes impuestos conforme los modelos se hacen más relajados.

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 programa 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, actúan 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.

6.4.4 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 red de interconexión no permiten las 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 de 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 hayan 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 lecturas 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 la dependencia entre los datos preservará 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 tiempos de respuesta en un multiprocesador en el caso de fallo en una lectura son grandes 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 decenas de ciclos, mientras que un fallo puede costar centenas de ciclos.

6.4.5 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.

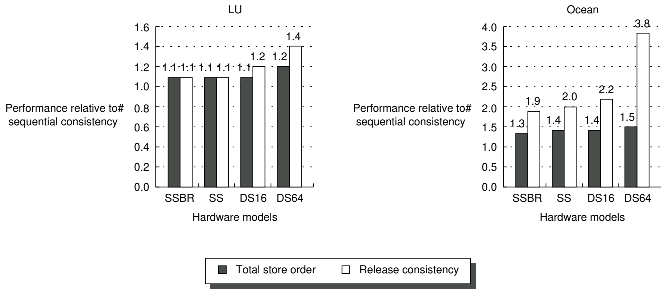


Figura 6.29: Rendimiento de los modelos de consistencia relajados sobre una variedad de mecanismos hardware.

La figura 6.29 muestra el rendimiento relativo para dos programa 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 significante de fallos de escritura. Al interpretar los datos de la figura 6.29 hay que recordar que las caché 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 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.

6.5 Diseño de protocolos *snoopy*

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 del 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 la caché 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 de 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 es el propietario y tiene la exclusividad. (La necesidad de distinguir propiedad de exclusividad será pronto clara.)

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 ser escrito en la memoria y puede ser descartado.

También debemos de 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.

6.5.1 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 *invalido* (I), *compartido* (S) y *modificado* (M). El estado de invalido tiene un significado claro. Compartido significa que el bloque está presente en la caché y no ha sido modificado, la memoria principal esta 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.

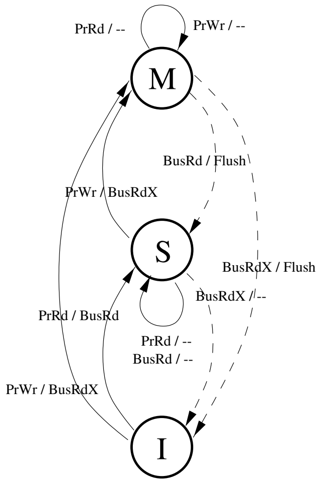


Figura 6.30: Protocolo básico de invalidación con tres estados.

El diagrama de estados que gobierna un bloque en cada caché en este protocolo se muestra en la figura 6.30. 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 invalido 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) eliminandolo 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 ultimo 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 6.21 se puede ver en la figura 6.31.

Acción Proces.	Est. en P_1	Est. en P_2	Estado en P_3	Acc. en el 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 6.31: Protocolo de invalidación de 3 estados en acción para las transacciones mostradas en la figura 6.21.

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 aparecerán 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 cual es la propia definición y como 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 6.32. 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 las transacciones de bus previas que genera, al mismo tiempo que sus propias escrituras están ordenadas por el propio programa.

También podemos ver como se cumple la CS en términos de la condiciones suficientes. La detección de la finalización de la escritura se realiza a través de 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 de 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

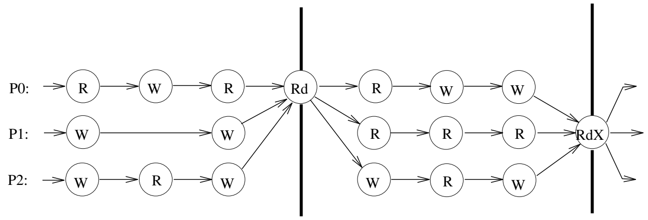


Figura 6.32: Orden parcial en las operaciones de memoria para una ejecución con el protocolo MSI.

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.

6.5.2 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 cargando 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 una 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.

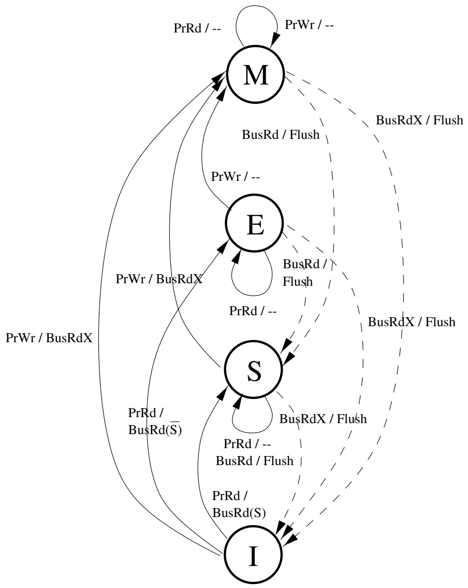


Figura 6.33: Diagrama de transición de estados para el protocolo Illinois MESI.

El diagrama de transición de estados para este protocolo se muestra en la figura 6.33. 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 BusRdX 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 son los mismos que en el caso del protocolo MSI.

6.5.3 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 (*Bus*WB*). 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 6.34 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

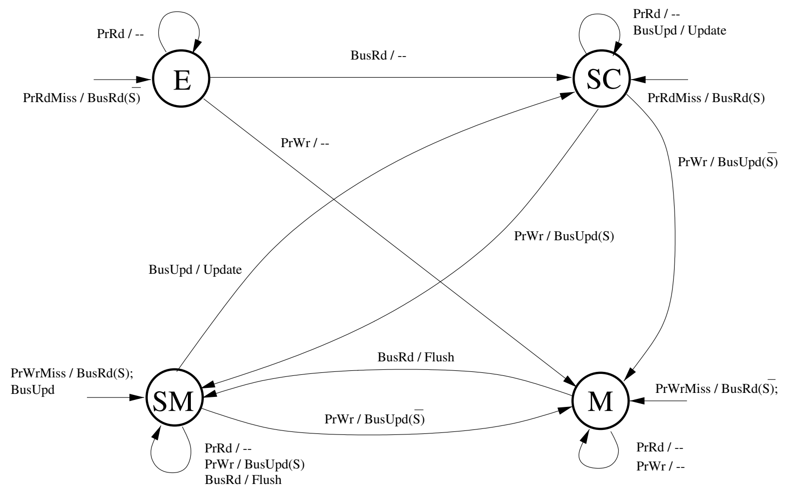


Figura 6.34: Diagrama de transición de estados del protocolo de actualización Dragon.

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

6.6 Rendimiento de los protocolos de coherencia *Snooping*

A la hora de determinar el rendimiento de un multiprocesador con arquitecturas como las vista hasta ahora, existen varios fenómenos que se combinan. En particular, el rendimiento total de la caché es una combinación del comportamiento del 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, tamaño del bloque puede afectar a estos dos componentes de diferente manera, llevando al sistema a un comportamiento que es una combinación de los dos efectos. En los siguientes apartados estudiaremos el rendimiento del protocolo MSI.

6.6.1 Rendimiento en programas paralelos del protocolo MSI

En este apartado usaremos un simulador para estudiar el rendimiento de cuatro programas paralelos utilizados frecuentemente como *benchmark*. En estas simulaciones el tamaño de los diferentes problemas serán:

- *Barnes-Hut* - 16K cuerpos durante seis iteraciones (el control de la exactitud esta fijado a 1.0, un valor típico y realista para este programa)
- *FFT* - 1 millón de datos complejos
- *LU* - Una matriz de 512×512 con bloques de 16×16
- *Ocean* - Una malla de 130×130 con una tolerancia de error típica

Al observar la tasa de fallos al variar el número de procesadores, tamaño de la caché y tamaño del bloque, descomponemos la tasa total de fallos en *fallos de coherencia* y fallos debidos a un uniprocesador normal. Los fallos de un uniprocesador normal son los debidos a la capacidad, existencia de conflictos y comportamiento compulsivo. Estos fallos los etiquetaremos como fallos de capacidad porque son la causa dominante en estos programas. En estas medidas, incluiremos como fallos de coherencia cualquier escritura necesaria para actualizar un bloque del estado compartido a exclusivo (modificado), incluso si nadie estaba compartiendo el bloque.

La figura 6.35 muestra las tasas de fallos para nuestras cuatro aplicaciones al incrementar el número de procesadores de 1 a 16, manteniendo el tamaño del problema constante. Al incrementar el número de procesadores, la caché total se incrementa, causando una caída de los fallos debidos a la capacidad. Por contra, este incremento causa un aumento en las comunicaciones entre las cachés, aumentando la tasa de fallos debido a la coherencia. La magnitud de estos dos efectos difieren según la aplicación.

En FFT, la tasa debida a la capacidad desciende (de cerca del 7% hasta el 5%) pero se incrementa la tasa debida a la coherencia (desde el 1% al 2.7%), lo que da lugar a una tasa total constante. Ocean muestra una combinación de efectos, incluyendo los debidos al particionamiento de la malla y como los bordes de la misma se asignan a los bloques de la caché. Para un código típico de malla 2D, las fallos generados por las comunicaciones son proporcionales al perímetro de cada partición de la malla, mientras que los fallos de capacidad son proporcionales al área de la malla. Por tanto, incrementar el tamaño total de la caché mientras que se mantiene el tamaño del problema fijo tendrá un efecto más significativo sobre la tasa de fallos debidos a la capacidad, al menos hasta que cada submalla se ajuste

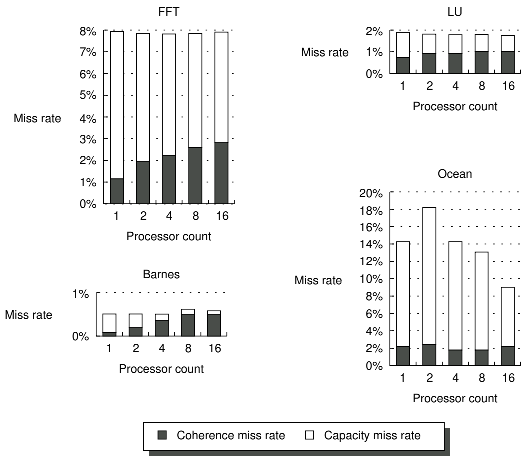


Figura 6.35: Tasa de fallos al variar el número de procesadores. Estos datos puede variar de manera no obvia al incrementarse el número de procesadores de 1 a 16.

en la caché del procesador. El salto entre la tasa de fallos para uno y dos procesadores se debe a los conflictos que surgen de la forma en que se asignan los elementos de la malla a la caché. Este conflicto está presente para las cachés de mapeo directo y las asociativas por conjuntos de dos vías (*two-way set associative cachés*), pero se desvanece para cachés con mayor asociatividad. Estos conflictos no son inusuales en las aplicaciones basadas en arrays, especialmente cuando se usan múltiples mallas a la vez. En Barnes y LU el incremento en el número de procesadores tiene poco efecto en la tasa de fallos, causando a veces un ligero incremento y a veces un ligero decrecimiento.

El incremento del tamaño de la caché tiene un efecto beneficioso sobre el rendimiento, ya que reduce la frecuencia de los fallos en la caché. La figura 6.36 ilustra el cambio en la tasa de fallos al aumentar el tamaño de la caché, mostrando la porción debida a fallos de coherencia y a fallos de capacidad. Dos efectos pueden conducir a que la tasa de fallos no decrezca -al menos no tan rápidamente como podríamos esperar- al incrementarse el tamaño de la caché: comunicacines inherentes y mesetas en la tasa de fallos. Las comunicaciones inherentes dan lugar a una cierta frecuencia en los fallos de cachés que no se ven afectados de forma significativa al incrementarse el tamaño de la caché. Además, si incrementamos el tamaño de la caché manteniendo el tamaño del problema fijo, la tasa de fallos debido a la coherencia limita el decrecimiento de la tasa de fallos de la caché. Este efecto es más obvio en Barnes, donde esta tasa es esencialmente la tasa de fallos total.

Un efecto menos importante es una meseta en la tasa de fallos de capacidad que aumenta cuando la aplicación tiene una fracción de sus datos presentes en la caché pero una porción significativa de los datos no caben en la caché o en cachés que son un poco mayores. En LU, una caché muy pequeña (aprox. 4 KB) puede albergar el par de bloques 16×16 que se usan en el bucle interior; mientras que el siguiente gran salto en la mejora ocurre cuando el tamaño total de la caché está entre 4 MB y 8 MB. Este efecto del *conjunto de trabajo* ocurre entre 32 KB y 128 KB para el caso de FFT, donde

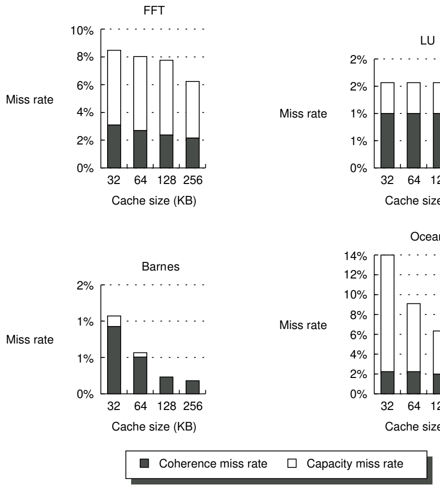


Figura 6.36: Efecto del incremento del tamaño de la caché en la tasa de fallos. El número de procesadores se mantiene fijo a 16.

la tasa de fallos debido a la capacidad baja únicamente un 0.3%. A partir de este tamaño de la caché, se observa un decrecimiento más rápido en la tasa de fallos, al empezar a residir en la memoria otras estructuras de datos importantes. Estas mesetas son comunes en programas que trabajan con grandes arrays de una manera estructurada.

Incrementar el tamaño del bloque es otra forma de cambiar la tasa de fallos en una caché. En los uniprocesadores, tamaños de bloque más grandes son a menudo óptimos con cachés más grandes. En los multiprocesadores, aparecen dos nuevos efectos: una reducción en la localidad espacial por la compartición de datos y un efecto denominado *falsa compartición*. Varios estudios han mostrado que los datos compartidos tienen menos localidad espacial que los datos no compartidos. Esto significa que para los datos compartidos, la carga de bloques mayores es menos efectiva que en un uniprocesador, ya que la probabilidad de que el bloque sea reemplazado antes de que todo su contenido sea referenciado es mayor.

El segundo efecto, la falsa compartición, surge del uso de algoritmos de coherencia basados en la invalidación con un único bit de válido por bloque. La falsa compartición ocurre cuando un bloque se invalida porque alguna palabra en el bloque, distinta de la que se esta leyendo, se ha modificado. Si la palabra que ha sido modificada se usa realmente en el procesador que ha recibido la invalidación, entonces la referencia era un compartición verdadera y habría causado un fallo independientemente del tamaño del bloque o de la posición de las palabras. Si, por el contrario, la palabra modificada y la palabra leída son diferentes y la invalidación no implica una comunicación del nuevo valor (protocolo basado en la actualización), sino únicamente un fallo de caché extra, entonces es un fallo debido a una falsa compartición. En una falsa compartición, el bloque está compartido, pero ninguna palabra

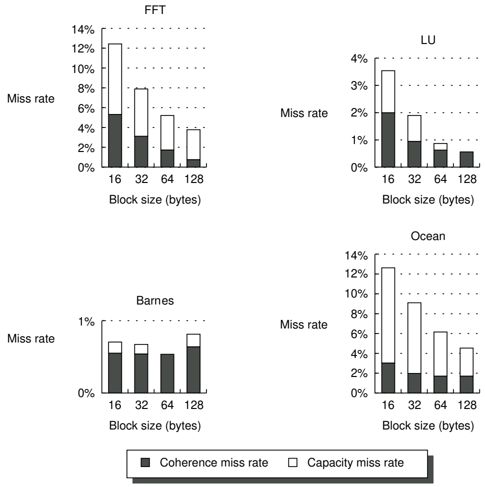


Figura 6.37: Tasa de fallos en función del aumento del tamaño del bloque. (16 procesadores, caché de 64 KB asociativa por conjunto de dos vías y tamaño de bloque de 32 bytes)

de la caché está realmente compartida, y el fallo de caché no hubiese ocurrido nunca si el tamaño del bloque fuera de una única palabra.

La figura 6.37 muestra las tasas de fallos al incrementarse el tamaño del bloque para una ejecución con 16 procesadores. El comportamiento más interesante ocurre en Barnes, donde la tasa de fallos inicialmente baja y después sube debido a un incremento en el número de fallos de coherencia, que probablemente ocurren debido a una falsa compartición. En los otros programas, el aumento del tamaño del bloque decrementa la tasa de fallos total. En Ocean y LU, el incremento del tamaño del bloque afecta de igual manera tanto a la tasa de fallos debido a la coherencia como la debida a la capacidad. En FFT, la tasa de fallos debido a la coherencia desciende más rápidamente que la debida a la capacidad. Esto es debido a que FFT está estructurado para ser muy eficiente. En programas menos optimizados, podríamos esperar una presencia mayor del problema de falsa compartición y una menor localidad espacial para los datos compartidos, dando lugar a un comportamiento más parecido al de Barnes.

Aunque la caída en las tasas de fallos con bloques grandes puede hacernos creer que la elección de bloques grandes es la mejor decisión, el cuello de botella en un sistema multiprocesador es a menudo la memoria y el ancho de banda del bus. Bloques mayores significan más bytes en el bus por cada fallo de caché. La figura 6.38 muestra el crecimiento en el tráfico del bus al incrementarse el tamaño del bloque. Este crecimiento es más serio en programas que tienen una tasa de fallos alta, especialmente Ocean. El crecimiento en el tráfico puede incluso llegar a producir caídas en el rendimiento debido a una penalización superior en el caso de fallos de caché y al incremento de la contención en el bus.

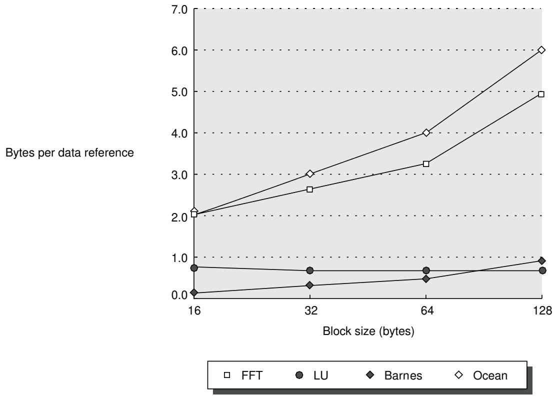


Figura 6.38: Tráfico en el bus debido a fallos de caché en función del tamaño del bloque en la caché de datos.

6.6.2 Rendimiento en multiprogramación

En este apartado examinaremos el rendimiento de la caché bajo una multiprogramación consistente en una actividad conjunta del usuario y el SO. En este caso consideraremos dos copias independientes de la fase de compilación del *Andrew Benchmark*. La fase de compilación consiste en un *make paralelo* usando hasta ocho procesadores. Dado la diferencia existente entre el comportamiento de los procesos del núcleo y los del usuario, las dos componentes se mantendrán separadas. Hay que recordar, sin embargo, que los procesos de usuario ejecutan más de ocho veces el número de instrucciones que los procesos del núcleo, así que la tasa de fallos total está determinada principalmente por la tasa de fallos del código de usuario, el cual, como veremos, es a menudo una quinta parte de la tasa de fallos del núcleo.

La figura 6.39 muestra la tasa de fallos de los datos en función del tamaño de la caché para el núcleo y los procesos de usuario. Los fallos se pueden dividir en tres grandes clases:

- Fallos compulsivos debidos al primer acceso a un bloque por parte de un procesador y que son significativos en este caso.
- Fallos de coherencia que representan fallos debidos a invalidaciones.
- Los fallos de capacidad incluyen los fallos causados por la interferencia entre los procesos de usuario y del SO y entre múltiples procesos de usuario. Los fallos debido a conflictos se han incluido en esta categoría.

Para esta carga de trabajo el comportamiento del sistema operativo es más complejo que el de los procesos de usuario. Esto se debe a dos razones. La primera es que el núcleo inicializa todas las páginas antes de asignarselas al usuario. La segunda razón es que el núcleo comparte datos y estos

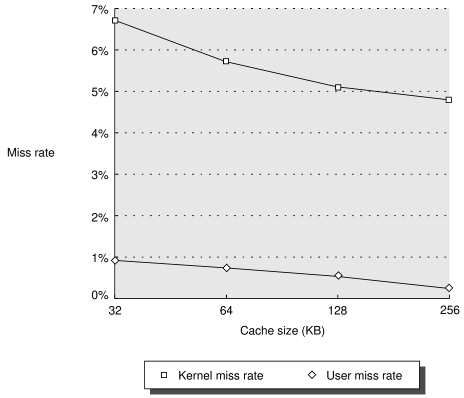


Figura 6.39: Tasa de fallos en función del tamaño de la caché. El tamaño del bloque es de 32 bytes.

presentan una tasa de fallos de coherencia no trivial. Por contra, los procesos de usuario sólo causan fallos de coherencia cuando el proceso se envía a ejecutar en un procesador diferente; este componente de la tasa de fallos es pequeño. La figura 6.40 muestra la disminución de la tasa de fallos del núcleo al incrementarse el tamaño de la caché.

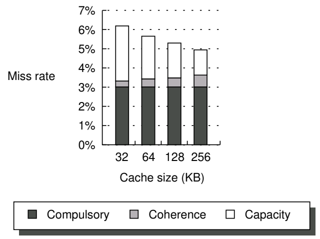


Figura 6.40: Componentes de la tasa de fallos de los datos del núcleo en función del tamaño de la caché.

El incremento del tamaño del bloque es probable que tenga efectos más beneficiosos bajo esta carga que en el caso de los programas paralelos, ya que una mayor parte de los fallos son compulsivos o debidos a la capacidad, y ambos pueden mejorar potencialmente al aumentar el tamaño del bloque. Dado que los fallos de coherencia son relativamente más raros, los efectos negativos de incrementar el tamaño del bloque deberían ser pequeños. La figura 6.41 muestra como varía la tasa de fallos para referencias del núcleo y de usuarios conforme aumenta el tamaño del bloque. La figura 6.42 confirma que, para referencias del núcleo, la reducción se debe sobre todo a la mejora en el caso de tasa de fallos compulsivos. Al igual que en el caso de cargas con programas paralelos, la ausencia de grandes incrementos en la tasa de fallos por coherencia al incrementarse el tamaño del bloque significa que los efecto de la falsa compartición son insignificantes.

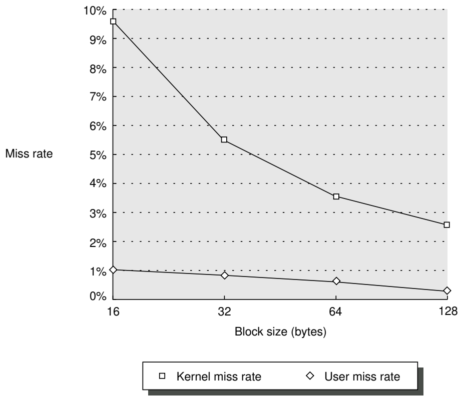


Figura 6.41: Tasa de fallos en función del tamaño del bloque para una caché de 32 KB asociativa por conjuntos de dos vías.

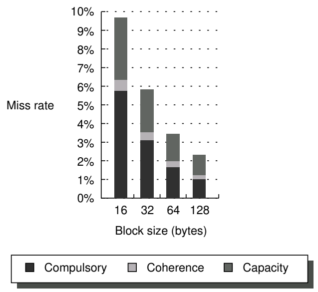


Figura 6.42: Descomposición de la tasa de fallos en función del tamaño del bloque.

Si examinamos el número de bytes necesarios por cada referencia de datos, como muestra la figura 6.43, podemos ver que el comportamiento de este tipo de carga es como el de algunos programas paralelos. La parte del núcleo presenta un ratio de tráfico mayor que crece rápidamente con el tamaño del bloque. Esto se produce a pesar de la significativa reducción en los fallos compulsivos; la menor reducción en los fallos de coherencia y de capacidad dan lugar a un incremento del trafico total. El programa de usuario tiene un tráfico mucho menor que crece muy lentamente.

Para cargas de multiprogramación, el SO es el que más peticiones realiza al sistema de memoria.

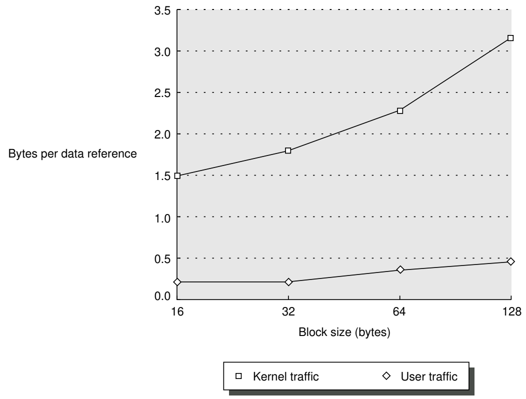


Figura 6.43: Bytes necesarios para cada referencia de datos para distintos tamaños de bloque.

Al incluir mayor actividad del SO en la carga de trabajo, la construcción de un sistema de memoria suficientemente capaz se convierte en muy difícil.

6.6.3 Resumen: Rendimiento de los esquemas de caché snoopy

En los apartados anteriores hemos examinado el rendimiento de la caché bajo dos supuestos: programas paralelos y entornos multiprogramados. Hemos visto que el tráfico debido a la coherencia puede introducir nuevos comportamientos en el sistema de memoria que no responden fácilmente a cambios en el tamaño de la caché o de los bloques que se usan normalmente para mejorar el rendimiento de la caché en sistemas uniprocesador. Las peticiones de coherencia son un componente significativo, pero no abrumador, en el caso del procesamiento paralelo. Sin embargo, es lógico esperar que las peticiones de coherencia sean más importantes en programas paralelos menos optimizados.

En entorno de multiprogramación, las porciones de usuario y SO se comportan de forma muy diferente, aunque ninguna de las dos tiene un trafico de coherencia significativo. En la parte del SO, las contribuciones compulsivas y de capacidad a la tasa de fallos son muy grandes, dando lugar a una tasa total que es comparable a los peores programas en la carga de trabajo paralela. Por otra parte, el rendimiento de la caché en la parte del usuario es muy buena y puede compararse con la de los mejores programas de la carga de trabajo paralela.

La pregunta de cómo estas tasas de fallos afectan al rendimiento de la CPU depende del resto del sistema de memoria, incluyendo la latencia y el ancho de banda del bus a la memoria.

6.7 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 aumentar 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 del 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, deben 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 debate 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* (LL) y una instrucción de escritura especial denominada *store conditional* (SC). 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 dicha operación se pudo 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 6.44 se muestra un ejemplo de como 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 6.44: 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*

6.7.1 Componentes de un evento de sincronización

Consideremos las componentes de un evento de sincronización, lo que nos permitirá ver claramente porqué 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 de 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 ha cambiado de valor. Una 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 las 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.

6.7.2 El papel del usuario, del sistema software y del hardware

¿Quién debe de 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 cuanto soporte hardware debe proporcionar y cuanta 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 hacen 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 normalmene con cargas de multiprogramación donde la planificación de los procesos y otras iteraciones 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.

6.7.3 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 existe 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.

6.7.3.1 Algoritmos simples para cerrojos

Considerad una operación de cerrojo (lock) usada para proporcionar atomicidad a una sección crítica. Para el método de adquisición, un proceso que intenta obtener el cerrojo debe chequear que el cerrojo está libre y si es así, hace suyo el cerrojo. El estado del cerrojo se puede almacenar en una variable binaria, donde 0 significa que está libre y 1 ocupado. En la operación de adquisición del cerrojo, un proceso que intenta obtener el cerrojo debería chequear si la variable es 0 y si es así, ponerla a 1. Si la variable es 1 (cerrojo ocupado), entonces debería esperar hasta que la variable vuelva a 0, usando el algoritmo de espera. Una operación unlock debería simplemente fijar la variable a 0 (método de liberación).


```
lock:   lw R2, 0(R1)
        bnez R2, lock
        li R2, #1
        sw R2, 0(R1)

unlock: li R2, #0
        sw R2, 0(R1)
```

El problema con este cerrojo, que se supone que proporciona atomicidad a la sección crítica que le sigue, es que necesita atomicidad en su propia implementación. Para ilustrarlo, vamos a suponer que la variable cerrojo estaba inicialmente fijada a 0 y dos procesos p_0 y p_1 ejecutan el código anterior. El proceso p_0 lee el valor de la variable cerrojo como 0 y piensa que está libre, asi que procede pasando la instrucción de salto. Su próximo paso es fijar la variable a 1, marcando el cerrojo como ocupado, pero antes puede suceder que el proceso p_1 lea la variable como 0, piense que está libre y también pasa la instrucción de salto. Por tanto, tenemos dos procesos que simultáneamente pasan el cerrojo y entran en la misma sección crítica, que era lo que el cerrojo intentaba evitar. La secuencia de las dos instrucciones, leer (testear) el cerrojo para chequear su estado y escribirlo (fijarlo) si está libre, no es atómico y no se puede prevenir que estas operaciones en distintos procesos se entremezclen. Lo que necesitamos es una forma de testear atómicamente el valor de una variable y fijarla a otro valor si el test tiene éxito, y después devolver si la secuencia atómica se ejecutó satisfactoriamente o no. Una forma para proporcionar esta atomicidad a los procesos de los usuarios es situar la rutina del cerrojo en el sistema operativo y acceder a él a través de una llamada al sistema, pero esto es caro y lleva a la pregunta de cómo el propio sistema operativo soporta los cerrojos. Otra opción es utilizar un cerrojo hardware, pero esta opción tiende a ser lenta en los procesadores modernos.

Una solución eficiente de uso general al problema de los cerrojos es soportar una instrucción atómica de lectura-modificación-escritura en el conjunto de instrucciones del procesador. Una solución típica es tener una instrucción de intercambio atómica: una posición de memoria se lee en un registro y otro valor se almacena en la posición. Existen variantes de esta operación con distintos grados de flexibilidad en la naturaleza del valor que se puede almacenar. Un ejemplo es la instrucción atómica *test&set:*el valor de la memoria se almacena en un registro específico y la constante 1 se almacena en la posición atómicamente. El éxito de *test&set* se determina examinando el valor del registro. Si es 0 el *test&set* ha tenido éxito. Se puede implementar un cerrojo (lock y unlock) con esta instrucción de la siguiente forma:

```
lock:   test&set R2, 0(R1)
        bnez R2, lock

unlock: li R2, #0
        sw R2, 0(R1)
```

Otra variante más sofisticada es por ejemplo la instrucción *swap*: lee el valor de una posición de memoria en un registro, pero en vez de escribir una constante fija en la posición de memoria, escribe el valor que había en el registro.

Otro ejemplo es la familia de instrucciones *fetch&op*: lee el valor actual de la posición en el registro y escribe el valor (que se ha obtenido aplicando la operación especificada por *fetch&op* al valor actual de la posición. La forma más simple es el *fetch&increment* y *fetch&decrement* que cambian el valor actual por 1. La instrucción *fetch&add* lleva otro operando, que es un registro o valor para sumar en el valor previo de la posición. Otra instrucción más compleja es *compare&swap*: coge dos registros y una posición de memoria, compara la posición de memoria con el contenido del primer registro y, si los dos son iguales, intercambia el contenido de la posición de memoria con el segundo registro.

6.7.3.2 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. Esta metas incluyen:

Baja latencia. Si un cerrojo está libre y ningún otro procesador está tratando de adquirirlo al mismo tiempo, un procesador debe de 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 escalar 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 de 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 de 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 es probable que el tiempo que se va a estar esperando es pequeño.

Una primera implementación de un cerrojo de este tipo sería la 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 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 la primera 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, modificando la implementación de tal manera que el bucle de 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 liberar el cerrojo, con lo que comienza de nuevo la carrera para adquirir el cerrojo (Test-and-Test&Set Lock). 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 como este esquema usa el mecanismo de coherencia de la caché. La figura 6.45 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 explicitamente 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)
        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.

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 6.45: 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.

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 como 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. 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 6.46 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 serán necesarios 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 6.46: 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 minimicen la serialización en el caso de que la contención sea significativa.

6.7.3.3 Algoritmos de cerrojos avanzados

Como ya se ha comentado es deseable tener un único proceso intentando obtener un cerrojo cuando este 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 6.47 muestra una implementación de este algoritmo. Esta implementación intenta preservar una baja latencia cuando la contención es pequeña no retrasando la primera iteración.

```
        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 6.47: Un cerrojo con *exponential back-off*.

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 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, este caso se discutirá en el capítulo 7.

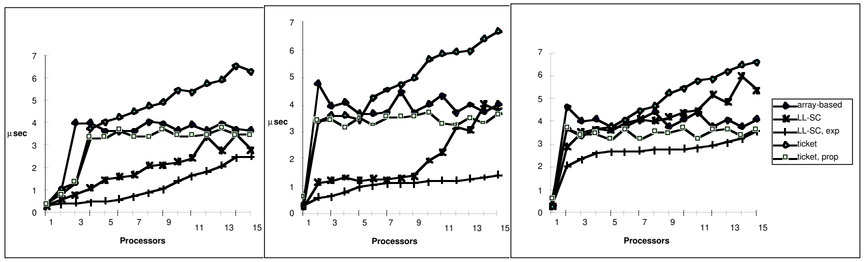
6.7.3.4 Rendimiento

A continuación examinaremos el rendimiento de diferentes cerrojos en el SGI Challenge, como se muestra en la figura 6.48. Todos los cerrojos se han implementado usando LL-SC, ya que Challenge únicamente proporciona estas instrucciones. El cerrojo *test&set* se ha implementado simulando la instrucción test&set mediante instrucciones 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.

Consideremos el caso de sección crítica nula. En este caso se observa que 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 incre-



(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 6.48: Rendimiento de los cerrojos en el SGI Challenge para tres posibles escenarios.

menta 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 será el que realice la lectura en primer lugar después de la liberación, de tal manera que el tiempo por transferencia no escala 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.

6.7.4 Eventos de sincronización punto a punto

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 flag separado, y evita la escritura y la lectura de la variable.

6.7.5 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.

6.7.5.1 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 dos cerrojos: uno usado para proteger un contador que lleva la cuenta de los procesos que llegan a la barrera y otro usado para bloquear a los procesos hasta que el último de ellos llega a la barrera. La figura 6.49 muestra una implementación típica de este tipo de barrera, denominado *barrera centralizada*.

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count = count +1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) { /* all arrived */
      count=0;/* reset counter */
      release=1;/* release processes */
}
else { /* more to come */

      spin (release=1);/* wait for arrivals */
}
```

Figura 6.49: 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 6.50 muestra el código para este tipo de barrera.

6.7.5.2 Rendimiento

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

Baja latencia (*longitud del camino crítico pequeña*). Ignorando la contención, nos gustaría que el número de operaciones dependientes necesarias para que p procesadores pasen la barrera sea pequeña.


```
local_sense = ! local_sense; /*toggle local_sense*/
lock (counterlock);/* ensure update atomic */
count++;/* count arrivals */
unlock (counterlock);/* unlock */
if (count==total) { /* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
else { /* more to come */
      spin (release=local_sense);/*wait for signal*/
}
```

Figura 6.50: Código para una barrera sense-reversing.

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 de ser pequeña y no debe escalar 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 6.51 se muestra el rendimiento de algunos algoritmos de barrera.

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

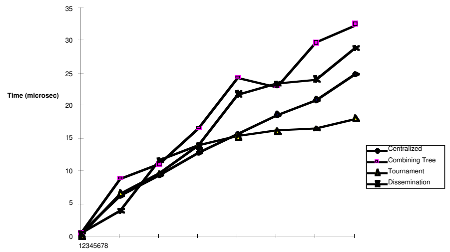


Figura 6.51: Rendimiento de algunas barreras en el SGI Challenge.

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 en el capítulo 7.

6.8 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, obteniendo una fracción mayor del tiempo de un procesador 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 todas 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, en donde procesadores y/o páginas de la memoria principal pueden ser reasignadas frecuentemente entre los diferentes procesos, como en lo físico, en 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 hacen 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 puedan utilizarlos como un recurso compartido o un servidor de computo 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.

Ejercicios

1. Considera el uso de un comutador de barras cruzadas de 8*x*8 para construir una red Omega multietapa de 512 entradas.

(a) ¿Cuántas etapas se requieren en la red?.

(b) ¿Cuántos conmutadores de barras cruzadas se necesitan para construir la red?

(c) Si la red se expande para que tenga 4096 nodos ¿Cuántos conmutadores adicionales de 8x8 se necesitan?.

2. Dibujar una red Omega de 16 entradas usando conmutadores de 2 *x* 2.

(a) Mostrar cómo se fijan los conmutadores para rutear un mensaje desde el nodo 1011 al nodo 0101 y desde el nodo 0111 al nodo 1001 simultáneamente. ¿Existe bloqueo?.

(b) Determinar cuántas permutaciones se pueden implementar en un paso a través de esta red Omega. ¿Cuál es el porcentaje de permutaciones de un paso entre todas las permutaciones?.

(c) ¿Cuál es número máximo de pasos necesarios para implementar cualquier permutación a través de la red?.

3. Considerar las conexiones del perfect-suffle (barajadura-2). Generalizar el mapeo para una barajadura-*m* sobre *n* objetos, donde *mxk=n* para algún entero $k \geq 2$, para la construcción de la clase de redes Delta (introducida por Patel, 1980).

(a) Mostrar cómo realizar una barajadura-4 sobre 12 objetos.

(b) Usar el mínimo número de conmutadores de 4 x 3 y barajadura-4 como patrón de conexión entre las etapas para construir una red Delta de 64 entradas y 27 salidas en tres etapas.

(c) ¿Cuál es la relación entre las redes Omega y Delta?.

4. ¿Está relacionado el problema de la coherencia de caches con los registros del procesador?. Dado que los registros no se mantienen consistentes en hardware, ¿cómo garantizan los sistemas actuales la semántica de los programas?.

5. Para cada una de las cadenas de referencias a memoria dadas, comparar el costo de ejecutarlas en una máquina basada en bus que soporta (a) el protocolo MESI y (b) el protocolo DRAGON. Explicar las diferencias de rendimiento observadas en términos de las características de las cadenas y de los protocolos de coherencia.

cadena 1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3

cadena 2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1

cadena 3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

Todas las referencias en las cadenas son a la misma posición: r/w indica lectura o escritura y el dígito se refiere al procesador que hace la referencia. Asume que todas las caches están inicialmente vacías, y usan el siguiente modelo de costo:

Un acierto de cache en lectura/escritura: 1 ciclo

Fallos que requieren una simple transacción del bus (BusUpgr, BusUpd): 60 ciclos,

Fallos que requieren que se transfiera todo el bloque de cache: 90 ciclos

6. Dados los siguienes fragmentos de código, indicar qué resultados son posibles (o no posibles) bajo la consistencia secuencial. Se asume que todas las variables están inicializadas a 0 antes de que el código sea ejecutado.

(a)

P_1	P_2	P_3
`A=1`	`u=A`	`v=B`
	`B=1`	`w=A`

(b)

P_1	P_2	P_3	P_4
`A=1`	`u=A`	`B=1`	`w=B`
	`v=B`		`x=A`

7. Considera un multiprocesador basado en bus con 4 procesadores que usa el protocolo MESI. Cada procesador ejecuta un lock *test&set* para conseguir el acceso a una sección crítica nula. Suponer que la instrucción *test&set* siempre que va por el bus necesita el mismo tiempo que una transacción normal de lectura. La condición inicial es que el procesador 1 tiene el lock y los procesadores 2, 3 y 4 están dando vueltas en sus caches esperando a que el lock sea liberado. Cada programa coge el lock una vez y sale del programa. Considerar sólo las transacciones del bus relacionadas con las operaciones de lock-unlock.

(a) ¿Cuál es el número mínimo de transacciones para ir del estado inicial al estado final?.

(b) ¿Cuál es el número de transacciones en el peor caso?.

(c) Repetir los apartados (a) y (b) suponiendo el protocolo Dragon.

8. Considerar el siguiente código paralelo que usa lock y unlock

```
parfor (i=0; i<n; i++)
{
  sección no crítica
   lock(S);
    sección crítica
    unlock(S);
}
```

Asume que la sección no crítica necesita T_{ncs}, la sección crítica T_{cs} y un lock(S) t_{lock} sg. El overhead del unlock(S) es despreciable. El código secuencial correspondiente necesita $n(T_{ncs} + T_{cs})$ sg para completarse. Se puede ignorar el overhead de paralelismo.

(a) ¿Cuál es el tiempo de ejecución paralelo total?

(b) ¿Cuál será el speedup al usar *n* procesadores?.

9. Supón que hay 20 procesadores en un bus que intentan un lock a una variable simultáneamente. Asume que cada transacción del bus (fallo de lectura o de escritura) necesita 50 ciclos como mucho. Se puede ignorar el tiempo de la lectura o escritura actual del lock en la cache, así como el tiempo para ejecutar otras operaciones que no sean de sincronización en la barrera.

(a) Determinar el número de transacciones del bus necesarias para que los 20 procesadores alcancen la barrera, sean liberados de la barrera, y salgan de la barrera.

(b) Asume que el bus es totalmente justo de tal manera que cualquier petición pendiente se sirve antes que cualquier nueva petición y que los procesadores son iguales de rápidos. ¿Cuánto tiempo llevará todo el proceso?.

10. Considerar el algoritmo de barreras *sense reversal*. ¿Habría algún problema si la sentencia UNLOCK hubiera sido situada justo después del incremento del contador en vez de después de cada salto de la condición *if*?. ¿Cuál sería?.