TEMA 2. JERARQUÍA DE MEMORIA

2.1. Introducción a las jerarquías de memoria
2.2. Nivel de memoria caché
2.3. Evaluación del rendimiento de una jerarquía de memoria
2.4. Memoria principal
2.5. Mejora del rendimiento de la memoria caché
2.6. Observaciones finales

2.1 INTRODUCCIÓN A LAS JERARQUÍAS DE MEMORIA

Los pioneros de los computadores predijeron correctamente que los programadores querrían cantidades ilimitadas de memoria rápida. Hoy en día existen programas que requieren 32 Mb de memoria como mínimo para funcionar, y bastante más del doble si queremos que su rendimiento sea aceptable. Con esto se comprueba que cada vez, los programas requieren mucha más memoria y ocupan un volumen de código mucho mayor. Además de este fenómeno, con la aparición de la multiprogramación, este efecto se ve aumentado de forma considerable, ya que los computadores han de tener una cantidad de memoria suficiente para almacenar varios programas (cuantos más mejor). Todo esto nos lleva a que las memorias han de tener una gran capacidad de almacenamiento.

Por otra parte, los computadores han visto aumentada la velocidad de la CPU. Es decir, que la CPU va a solicitar datos de la memoria con una frecuencia mucho mayor que antes. Esto conlleva que las memorias actuales han de ser mucho más rápidas.

A la hora de implementar las memorias, los dos conceptos de gran capacidad de almacenamiento y de rapidez son contrapuestos. Aunque, realmente el problema surge principalmente por causas económicas: la memoria, cuanto más rápida y más grande, más cara. Por lo tanto, hay que idear un sistema para aparentar una memoria con dichas características a partir de memorias más baratas.

EL PRINCIPIO DE LOCALIDAD

Como predice la ***regla 90/10*** *(un programa emplea el 90% del tiempo en la ejecución del 10% del código)*, la mayoría de los programas, afortunadamente, no acceden a todo el código o a los datos de memoria uniformemente. Esta regla se puede plantear como el ***Principio de Localidad*** *(todos los programas favorecen una parte de su espacio de direcciones en cualquier instante de tiempo)*. Tiene dos dimensiones:

- *Localidad temporal* (localidad en el tiempo) $\Rightarrow$ Si se referencia un elemento, tenderá a ser referenciado pronto.
- *Localidad espacial* (localidad en el espacio) $\Rightarrow$ Si se referencia un elemento, los elementos cercanos a él tenderán a ser referenciados pronto.

Una *jerarquía de memoria* es una reacción natural a la localidad y tecnología. Gracias al *Principio de Localidad*, se puede escalonar la memoria en niveles. El nivel más cercano al procesador debe ser rápido y puede ser reducido; en el siguiente nivel, la memoria no tiene por qué ser tan rápido, ya que accederemos menos veces, y por razones de coste puede ser más amplia; y así, hasta formar toda una *jerarquía de memoria*. Normalmente hablaremos de las gestiones entre dos niveles adyacentes, el superior y el inferior, aunque la memoria posea multitud de ellos. Definiremos como *nivel superior* entre dos niveles de memoria al más cercano a la CPU (Figura 2.1). Todos los datos de un nivel superior están contenidos en el nivel inferior siguiente, sucesivamente.

En cada nivel de la *jerarquía* encontramos estructuras de información concretas. A la unidad mínima que puede estar presente en la jerarquía de dos niveles se denomina ***bloque***. El tamaño de un bloque puede ser fijo o variable; si es fijo, el tamaño de memoria es un múltiplo de ese tamaño de bloque.

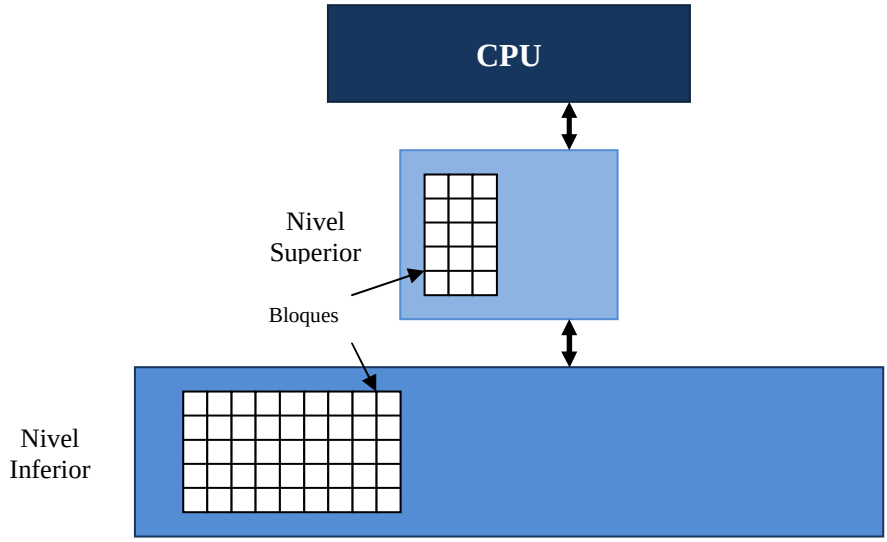


Figura 2.1 Par de niveles de jerarquía de memoria (Superior e Inferior).

El éxito o fracaso de un acceso al nivel superior se designa como *acierto* o *fallo*:

- *Acierto (hit)* ⇒ Acceso a una información de memoria que se encuentra en el nivel superior (está presente el bloque que se busca).
- *Fallo (miss)* ⇒ Acceso a una información de memoria que no se encuentra en el nivel superior (no está presente el bloque que se busca).

- *Frecuencia o tasa de aciertos* ⇒ Fracción de accesos a memoria encontrados en el nivel superior (a veces se representa en %).
- *Frecuencia o tasa de fallos* ⇒ Fracción de accesos a memoria no encontrados en el nivel superior ⇒ (1 – *frecuencia de aciertos*).

- *Tiempo de acierto* ⇒ Tiempo para acceder al nivel superior de la jerarquía de memoria. Incluye el tiempo para determinar si el acceso es un acierto o un fallo.
- *Penalización de fallo* ⇒ Tiempo para sustituir un bloque del nivel superior por el bloque correspondiente del nivel más bajo. La *penalización de fallo* se divide a su vez en:

    - *Tiempo de acceso* ⇒ Tiempo para acceder a la primera palabra de un bloque en un fallo. (Está relacionado con la latencia del nivel más bajo de memoria y a veces se denomina *latencia de acceso*).
    - *Tiempo de transferencia* ⇒ Tiempo adicional para transferir las restantes palabras de bloque. (Está relacionado con el ancho de banda entre las memorias de nivel superior y nivel inferior).

La *jerarquía de memoria* nos lleva a una división de la **dirección de memoria**, donde encontramos el bloque en el nivel de la jerarquía y un indicador del elemento en el bloque. La *dirección de la estructura del bloque* es la parte de orden superior de la dirección, que identifica un bloque en ese nivel de la jerarquía. La *dirección del desplazamiento del bloque* es la parte de orden inferior de la dirección e identifica un elemento en un bloque (su número de bits es el log_2 *del tamaño de bloque*). (Figura 2.2).

Dirección de la estructura de bloque	*Dirección del desplazamiento de bloque*
10100001010101111110000	111001110

Figura 2.2 Ejemplo de las partes de dirección de la estructura y dirección del desplazamiento de una dirección de memoria del nivel inferior de 32 bits. El tamaño del bloque es de 2^9 = 512. El tamaño de la dirección de la estructura del bloque es de 23 bits (32 – 9).

Esta estructuración nos complica los accesos a memoria. Los procesadores sin jerarquía de memoria son mucho más simples ya que no necesitan saber manejar **tiempos de acceso a memoria variables**. Además, si la *penalización en los fallos* es muy elevada, es necesario un tratamiento de interrupciones para liberar el procesador de la espera, con el almacenamiento de todos los datos antes del fallo. El procesador también necesita un mecanismo para determinar si la información está en el nivel inferior de la jerarquía, lo que supone un gasto de tiempo adicional en cada acceso. En el caso de fallo, el procesador (hardware adicional) deberá ser capaz de transferir un bloque entre un nivel superior y uno inferior.

2.2 NIVEL DE MEMORIA CACHÉ

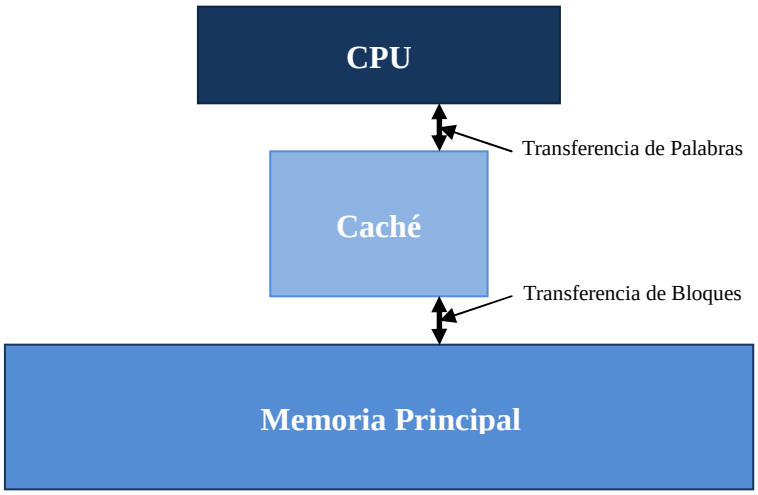


Figura 2.3 Memoria Caché y Memoria Principal.

En cada ciclo de instrucción, el procesador debe acceder a memoria al menos una vez para buscar la instrucción, y frecuentemente realiza múltiples accesos para leer los operandos o almacenar los resultados. Así, los ciclos para los accesos a memoria son una seria limitación ante la diferencia de velocidades entre el procesador y la memoria. La solución más usada históricamente ha sido el incremento del número de registros, pero es una solución cara. La solución será buscar una memoria, denominada *caché*, más pequeña y rápida que la *memoria principal* y que se sitúa entre ésta y el procesador. Este concepto se muestra en la Figura 2.3.

Por lo tanto, hay una *memoria principal* relativamente grande y lenta, conjuntamente con una *memoria caché* pequeña y rápida. Con esta estructura se utiliza de forma inteligente el *principio de localidad*.

La *memoria caché* almacena una copia de ciertas partes de la *memoria principal*. Cuando la *CPU* intenta leer una palabra de memoria, en primer lugar comprueba si la palabra deseada está ya en la *memoria caché*; si está, se lee la palabra desde la *memoria caché*; si no está, se transfiere a la *memoria caché* un **bloque** de la *memoria principal*, que contiene un determinado número de palabras. Debido al fenómeno de la localidad de las referencias a memoria, es probable que cuando se transfiere un bloque de datos a la *memoria caché*, las futuras llamadas a memoria se hagan a otras palabras contenidas en el bloque transferido.

En la Figura 2.4 se muestra la estructura de un *sistema* M_p-M_{ca}. La *memoria principal* consta de 2^n palabras, y cada palabra se puede referenciar mediante una dirección única de n bits.

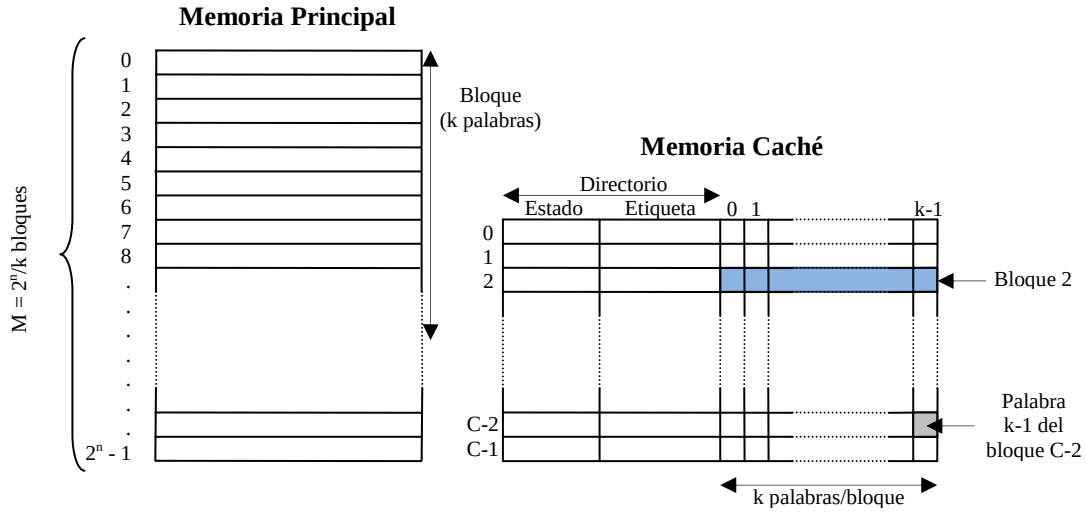


Figura 2.4 Estructura Memoria Principal (M_p) – Memoria Caché (M_{ca}).

Con el fin de efectuar la transformación entre la *memoria principal* y la *memoria caché*, se considera que la *memoria principal* está constituida por una serie de bloques de longitud fija, de k palabras por bloque. Hay $M = 2^n/k$ bloques en la *memoria principal*. La *memoria caché* contiene C bloques de k palabras cada uno, y el número de bloques de la *memoria caché* es considerablemente menor que el número de bloques que hay en la *memoria principal* ($C <<< M$). Esto implica que la *memoria caché* tiene que disponer de un ***directorio*** que consta de ***etiquetas*** que permiten identificar qué bloque de la *memoria principal* se encuentra en cada momento en la *memoria caché*. Los ***bits de estado*** sirven para indicar, entre otras cosas, si la información que contiene un bloque de caché es válida o no.

Para analizar cómo se realiza en general la operación de lectura en una *memoria caché*, se emplea la Figura 2.5.

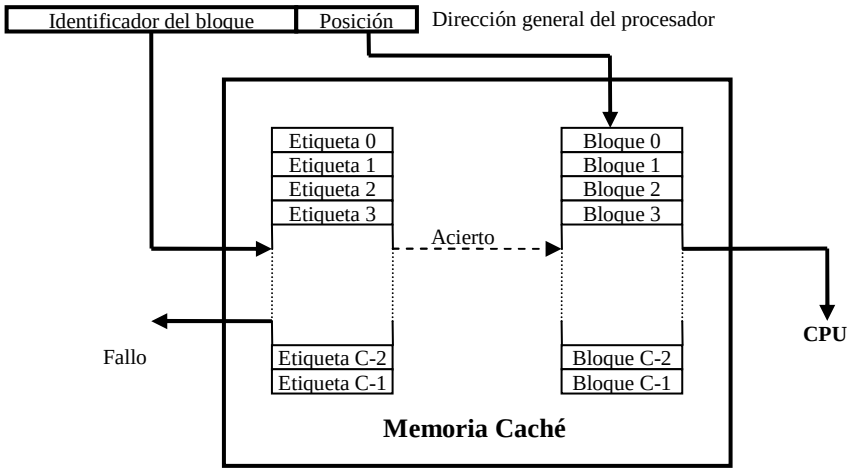


Figura 2.5 Acceso a la Memoria Caché (M_{ca}) en una operación de lectura.

La CPU genera una dirección para llevar a cabo un acceso a una determinada palabra de la memoria principal. Esta dirección, desde el punto de vista de la memoria caché, consta de dos partes: *identificador de bloque*, que especifica el bloque donde se encuentra la palabra pedida, y la otra indica la *posición* de la palabra dentro del bloque. Al realizar la búsqueda pueden ocurrir dos situaciones:

1. Que el *identificador de bloque* buscado coincida con alguna de las etiquetas de los bloques almacenados en el directorio de la memoria caché $\rightarrow$ Se ha producido un ***acierto***, y con los bits correspondientes al campo de *posición* se selecciona la palabra pedida dentro del bloque para entregársela a la CPU.
2. Que el *identificador del bloque* buscado no coincida con alguna de las etiquetas de los bloques almacenados en el directorio de la caché $\rightarrow$ Se ha producido un ***fallo***, la palabra pedida no se encuentra en ese momento en la memoria caché. En este caso, para realizar la operación de lectura solicitada por el procesador es necesario acceder a la memoria principal; debiéndose actualizar la memoria caché para que la próxima vez, tanto si se hace referencia a esa dirección como a otra próxima, no se produzca fallo (por el *Principio de Localidad*). Generalmente, la actualización lleva consigo la sustitución de uno de los bloques de la memoria caché así como la modificación de la etiqueta correspondiente.

La Figura 2.6 muestra el diagrama de flujo de la operación de lectura en la memoria caché.

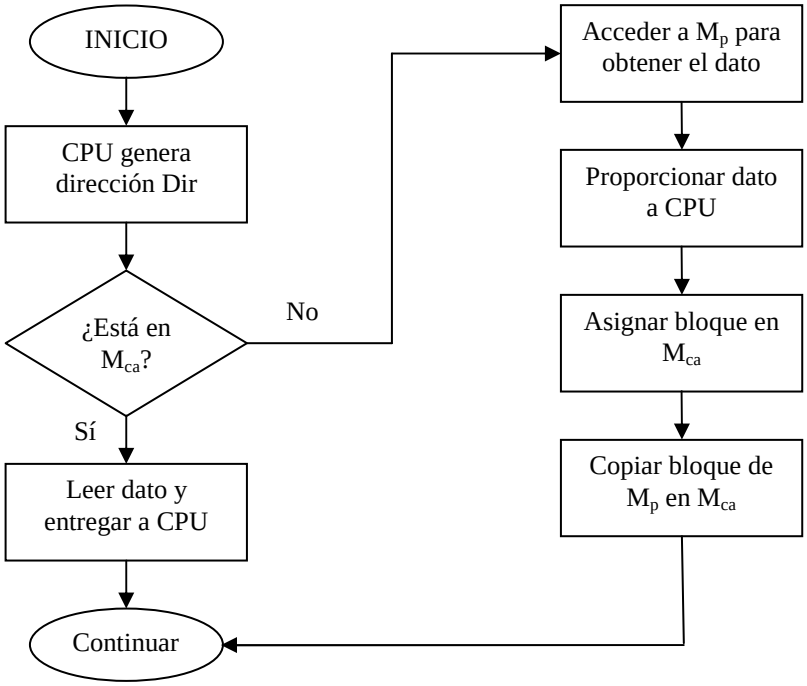


Figura 2.6 Diagrama de flujo de la operación de lectura de una Memoria Caché (M_{ca}).

Aunque hay gran diversidad de memorias caché, existen unos ciertos criterios básicos de diseño que sirven para clasificar y diferenciar las distintas arquitecturas de este tipo de memorias:

1. ***Capacidad***:

   - 1K
   - 4K
   - 16K
   - 32K
   - 64K
   - 128K
   - 256K

   .
   .

2. ***Organización de la memoria caché***:

   - Directa
   - Totalmente asociativa
   - Asociativa por conjuntos

3. ***Mecanismos de búsqueda o políticas de búsqueda***:

   - Por demanda
   - Con anticipación
   - Selectiva

4. ***Algoritmo de reemplazamiento***:

    - LRU (Least Recently Used – Utilizado menos recientemente)
    - FIFO (Fist In First Out – Más antiguo)
    - LFU (Least Frecuently Used – Utilizado menos frecuentemente)
    - Aleatorio

5. ***Estrategia de escritura o política de actualización***:

    - Escritura inmediata
    - Post-escritura

6. ***Tamaño del bloque***:

    - 4 palabras
    - 8 palabras
    - 16 palabras
    - 32 palabras

    .
    .

7. ***Número de cachés***:

    - Cachés de un nivel
    - Cachés de dos niveles

    .
    .

1. CAPACIDAD DE LA MEMORIA CACHÉ

El tamaño de la memoria caché (número de palabras) plantea un cierto compromiso. Por un lado debería ser lo suficientemente pequeña como para que el coste medio por bit de información almacenado en la *memoria interna* del computador estuviese próximo al de la memoria principal. Por otro lado tendría que ser lo bastante grande como para que el tiempo de acceso medio total fuese lo más próximo posible al de la memoria caché. No obstante, hay algunas otras motivaciones para tratar de minimizar el tamaño de una memoria caché; cuanto más grande sea la memoria caché, más compleja será la lógica asociada con su direccionamiento; el resultado neto es que la memoria caché de gran capacidad tienden a ser más lentas que las más pequeñas, incluso empleando en ambas la misma tecnología de fabricación de circuitos integrados. También su capacidad tiende a estar limitada por el espacio físico que se le asigna, en muchos casos dentro del mismo circuito integrado que el procesador.

2. ORGANIZACIÓN DE LA MEMORIA CACHÉ

Existen diversas formas para organizar internamente una caché para almacenar su información. En todos los casos, la CPU referencia a la caché con la dirección de memoria principal del dato que se necesita (la CPU siempre se dirige al espacio de memoria principal). Por lo tanto, cualquier organización de memoria caché debe utilizar esta dirección para encontrar el dato si lo tiene almacenado o indicar cuando ha ocurrido un fallo. La forma de hacer corresponder la información mantenida en la memoria principal con su equivalente en la caché debe de implementarse totalmente en hardware si se desea conseguir un rendimiento óptimo en la operación del sistema. Básicamente, *la organización de la memoria caché consiste en establecer la función*

de correspondencia que asigna a los bloques de la memoria principal posiciones definidas en la memoria caché. Para el cálculo de dicha función de correspondencia se emplean tres técnicas básicas:

- Directa
- Totalmente asociativa
- Asociativa por conjuntos

Correspondencia directa

Es la técnica más simple de todas. Cada bloque de la memoria principal se transforma en un único bloque de la memoria caché. Se muestra en la Figura 2.7.

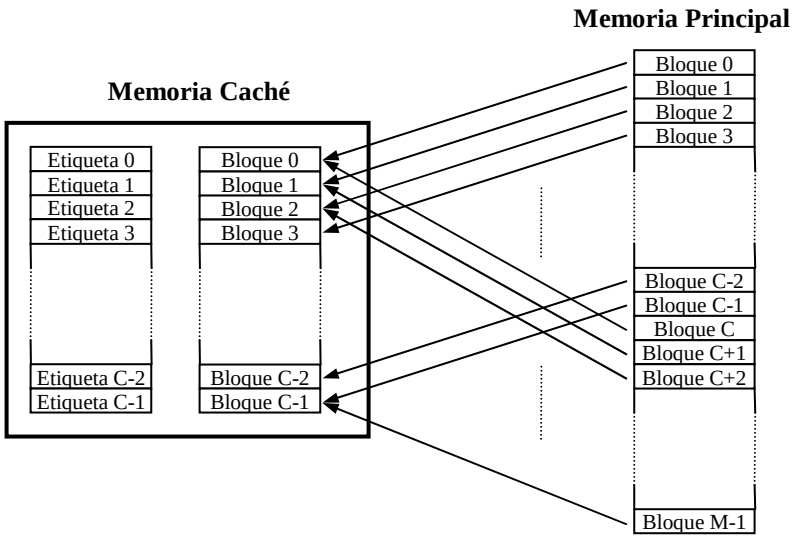


Figura 2.7 Asignación de bloques de la M_p en la M_{ca} con correspondencia directa.

La función de transformación es:

$$i = j \text{ módulo } C$$

i = Número de bloque asignado en la M_{ca} al bloque de la M_p

j = Número de bloque de la M_p

C = Número de bloques que tiene la M_{ca}

La Figura 2.8 ilustra el mecanismo general de la correspondencia directa.

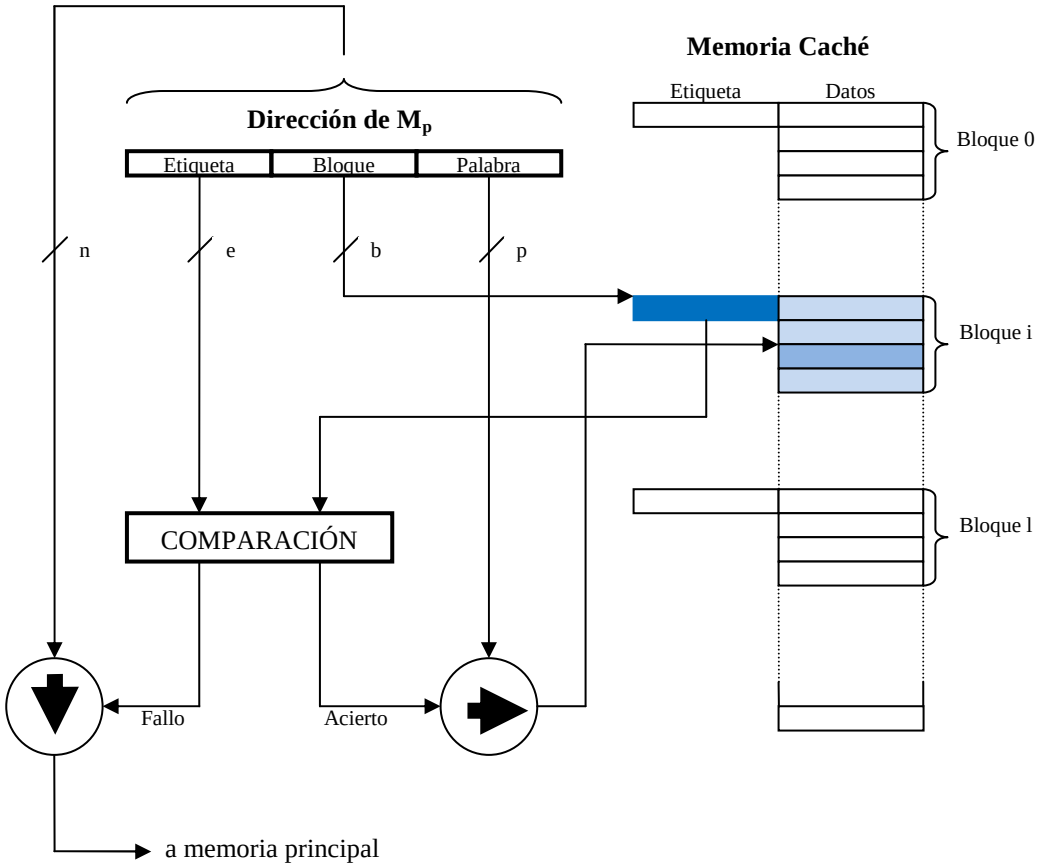


Figura 2.8 Organización de una memoria caché con correspondencia directa.

Cuando la CPU genera una dirección (n bits) para acceder a una palabra de la memoria principal, su formato desde el punto de vista de la memoria caché se divide en tres campos: *etiqueta*, *bloque* y *palabra*. El número de bits de cada uno de estos campos viene dado por las relaciones siguientes:

$$\textit{Campo palabra:}\ p = \log_2 (k) \qquad \textit{Campo bloque:}\ b = \log_2 (C)$$

$$\textit{Campo etiqueta:}\ e = n - p - b$$

n = Número de bits de una dirección de M_p

p = Número de bits para seleccionar una palabra de un bloque

k = Número de palabras/bloque

b = Número de bits para seleccionar un bloque en la M_{ca}

C = Número de bloques que tiene la M_{ca}

e = Número de bits que definen la etiqueta

El funcionamiento de la correspondencia directa se puede describir de la siguiente forma:

1) Con el campo *bloque* se identifica un bloque de la M_{ca}.

2) Se compara la etiqueta de ese bloque de la M_{ca} con el campo *etiqueta* de la dirección solicitada por el procesador. Si coinciden, entonces con el campo *palabra* se selecciona la palabra pedida dentro del bloque y se le entrega a la CPU; si no coinciden las etiquetas, quiere decir que no se encuentra en la M_{ca} y por lo tanto se produce un fallo y habrá que ir a buscar la palabra a la M_p.

La técnica de correspondencia directa es simple y poco costosa de realizar. Su principal desventaja es que cualquier bloque dado tiene asignada una posición fija en la memoria caché. Así, si ocurre que un programa efectúa repetidas referencias a palabras de dos bloques diferentes de la M_p que tienen asignado el mismo bloque de la M_{ca}, estos bloques se estarán moviendo continuamente entre la memoria caché y la memoria principal, con la consiguiente pérdida de rendimiento del sistema.

Correspondencia totalmente asociativa

Esta técnica subsana el inconveniente que se acaba de mencionar al estudiar la correspondencia directa, ya que permite que un bloque de memoria principal se cargue en cualquier bloque de la memoria caché, tal como se muestra en la Figura 2.9.

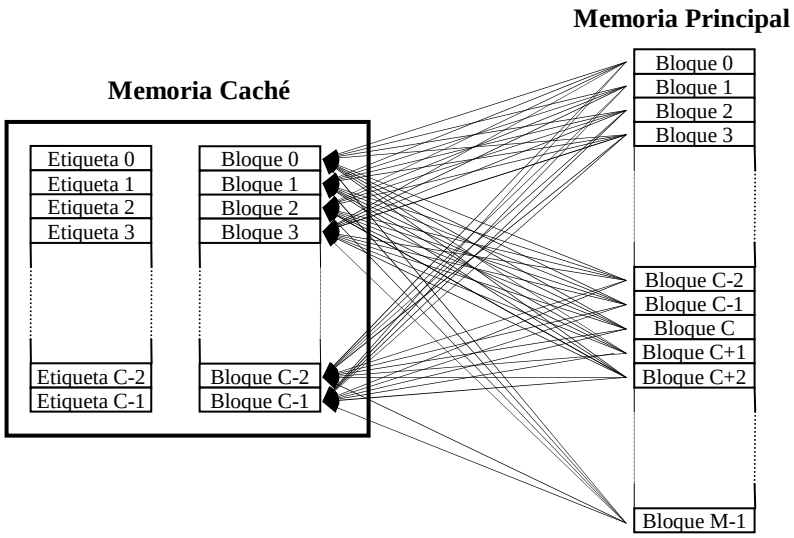


Figura 2.9 Asignación de bloques de la M_p en la M_{ca} con correspondencia totalmente asociativa.

Cuando la CPU genera una dirección para acceder a una palabra de la memoria principal, su formato desde el punto de vista de la memoria caché se divide en dos campos: *etiqueta* y *palabra*. El número de bits de cada uno de estos campos viene dado por las relaciones siguientes:

Campo palabra: $p = \log_2 (k)$	*Campo etiqueta:* $e = n - p$

n = Número de bits de una dirección de M_p

p = Número de bits para seleccionar una palabra de un bloque

k = Número de palabras/bloque

e = Número de bits que definen la etiqueta

La Figura 2.10 ilustra el mecanismo general de la correspondencia totalmente asociativa.

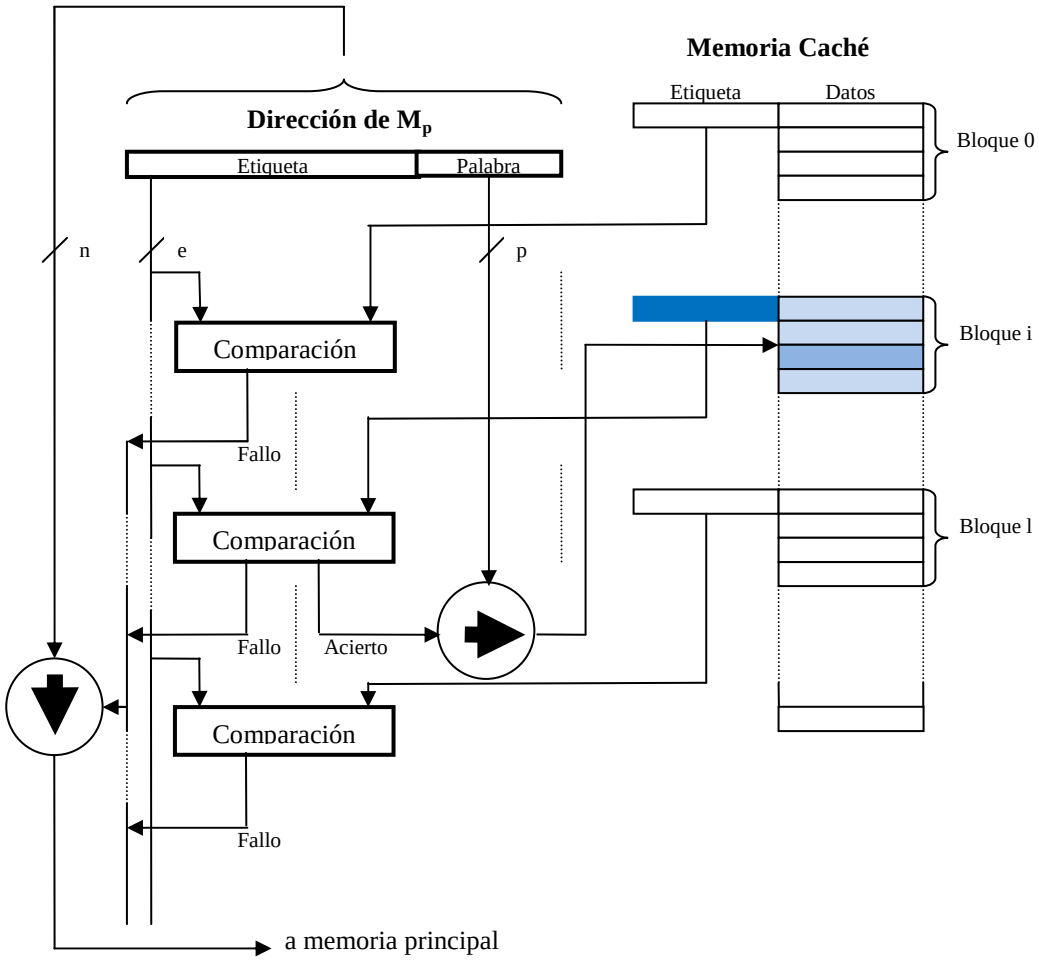


Figura 2.10 Organización de una memoria caché con correspondencia totalmente asociativa.

El funcionamiento de la correspondencia totalmente asociativa se puede describir en los términos que siguen:

1) Para determinar si un bloque está ya en la memoria caché es preciso que ésta incorpore la lógica necesaria que le permita examinar de forma simultánea todas las etiquetas de su directorio y comprobar si el campo *etiqueta* de la dirección solicitada por el procesador coincide con alguna de ellas. Un bloque de memoria principal se puede almacenar en cualquier bloque de la memoria caché, a condición de que se almacene con él los “e bits” de su campo de *etiqueta*.
2) Si coinciden, con el campo *palabra* se selecciona la palabra pedida dentro del bloque y se le entrega a la CPU. Si al comparar las etiquetas no coinciden quiere decir que no se encuentra en la memoria caché y por lo tanto se produce un fallo y habrá que ir a buscar la palabra a memoria principal.

Cuando se lee un nuevo bloque, con la correspondencia totalmente asociativa, hay que decidir generalmente por cuál se va a sustituir en la memoria caché. Los algoritmos de reemplazamiento se diseñan con el fin de optimizar la probabilidad de encontrar la palabra solicitada en la memoria caché. La principal desventaja de este procedimiento es la necesidad de una circuitería, bastante compleja, para examinar en

paralelo los campos de etiqueta de todos los bloques de la memoria caché (aumenta la complejidad de acceso). Aunque tiene como ventaja el que dos bloques con localidad de referencia temporal no competirán por la misma línea (bloque en la caché).

Correspondencia asociativa por conjuntos

La técnica de *correspondencia asociativa por conjuntos* es un compromiso que trata de aunar las ventajas de los dos métodos anteriores ⇒ La memoria caché se divide en ***q conjuntos***, cada uno de los cuales consta de ***r bloques***:

$$C = q \times r$$
$$i = j \text{ módulo } q$$

C = Número de bloques que contiene la M_{ca}

q = Número de conjuntos en los que se divide la M_{ca}

r = Número de bloques que contiene cada uno de los conjuntos

i = Número de conjunto asignado en la M_{ca}

j = Número de bloque de la M_p

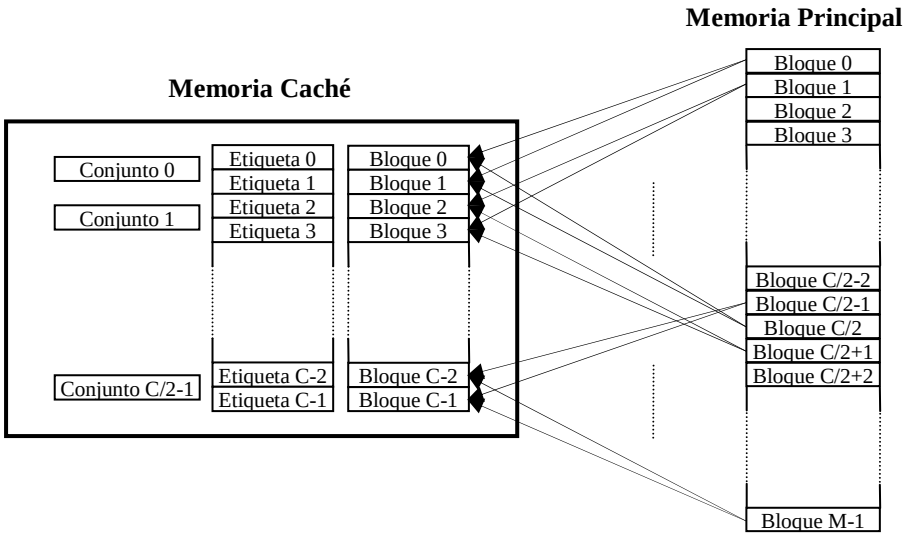


Figura 2.11 Asignación de bloques de la M_p en la M_{ca} con correspondencia asociativa por conjuntos.

Cada *bloque* de la memoria principal puede ubicarse en uno de los ***r bloques*** de la memoria caché que componen cada conjunto ⇒ r define el ***grado de asociatividad*** (o ***número de vías***) de la memoria caché. Con este algoritmo, el bloque j de la memoria principal se puede almacenar en un bloque cualquiera del conjunto que tiene asociado en la memoria caché. En la Figura 2.11 se muestra un ejemplo para el caso en que el número de bloques que contiene un conjunto r = 2; por lo que el número de conjuntos en los que se divide la M_{ca} será q = C/2.

La correspondencia asociativa por conjuntos se puede ver como una correspondencia directa entre los bloques de la M_p y los conjuntos de la M_{ca}, y como una correspondencia totalmente asociativa entre los bloques de un mismo conjunto.

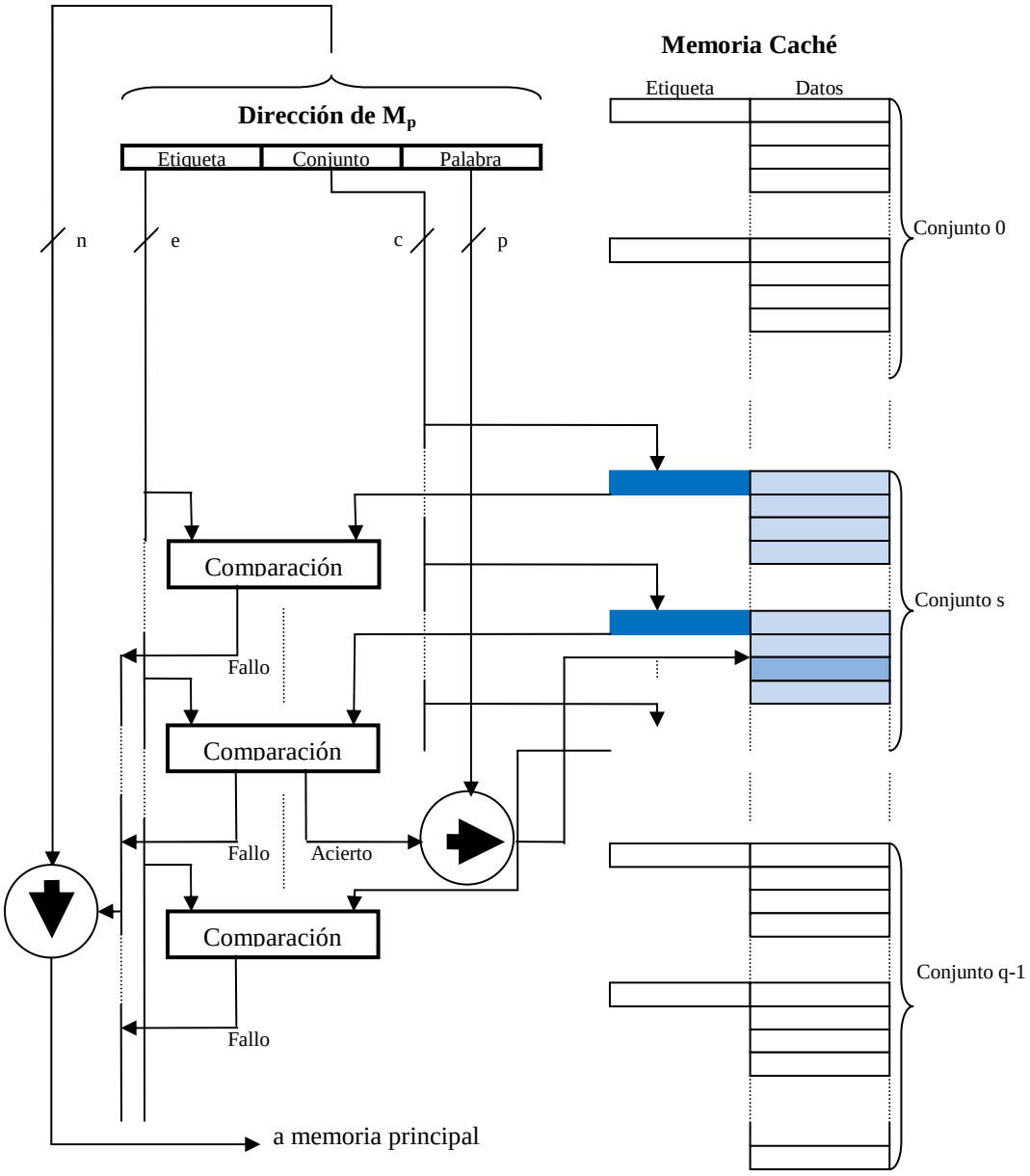


Figura 2.12 Organización de una memoria caché con correspondencia asociativa por conjuntos.

La Figura 2.12 ilustra el mecanismo general de la correspondencia asociativa por conjuntos. Cuando la CPU genera una dirección para acceder a una palabra de la M_p, su formato desde el punto de vista de la M_{ca} se divide en tres campos: *etiqueta, conjunto y palabra*. El número de bits de cada uno de estos campos viene dado por las relaciones siguientes:

Campo palabra: $p = \log_2 (k)$

Campo conjunto: $c = \log_2 (q)$

Campo etiqueta: $e = n - p - c$

n = Número de bits de una dirección de M_p

p = Número de bits para seleccionar una palabra de un bloque

k = Número de palabras/bloque

c = Número de bits que definen el conjunto

q = Número de conjuntos en los que se divide la M_{ca}

e = Número de bits que definen la etiqueta

Los c bits del campo conjunto especifican uno de entre $q = 2^c$ conjuntos en los que se divide la M_{ca}. Los (e + c) bits de los campos de etiqueta y conjunto especifican uno de los $M = 2^{n-p}$ bloques de la M_p. Cada bloque en cada conjunto tiene una etiqueta almacenada que conjuntamente con el número del conjunto completa la identificación del bloque de la M_{ca}. El funcionamiento de la correspondencia asociativa por conjuntos se puede describir en los términos siguientes:

1) En primer lugar se utiliza el número de conjunto de la dirección solicitada por el procesador para acceder al conjunto correspondiente de la M_{ca}.
2) Se comparan simultáneamente todas las etiquetas del conjunto seleccionado con la etiqueta de la dirección solicitada. Si coinciden entonces se accede a la palabra pedida en la M_{ca}, en caso contrario se produce un fallo y habrá que ir a buscar la palabra a M_p.

La correspondencia asociativa por conjuntos contiene las dos técnicas anteriores (directa y totalmente asociativa) como casos particulares:

a) q = C, r = 1 $\Rightarrow$ Correspondencia directa.
b) q = 1, r = C $\Rightarrow$ Correspondencia totalmente asociativa.

La utilización de q = C/2, r = 2 (es decir, 2 bloques por conjunto), es la organización más usual en la correspondencia asociativa por conjuntos. En este caso se mejora significativamente el rendimiento de la caché frente a la transformación directa. También se emplea q = C/4, r = 4 (es decir, 4 bloques por conjunto) que proporciona una relativa mejora a un coste moderado. A partir de este valor, el beneficio marginal que se obtiene al aumentar el número de particiones por conjunto no compensa, por el mayor coste que supone.

3. MECANISMOS DE BÚSQUEDA O DE EXTRACCIÓN

Los *mecanismos de búsqueda* o *políticas de búsqueda* determinan la causa que desencadena la llevada de un bloque a la memoria caché (normalmente un fallo en la referencia) $\Rightarrow$ Determinan las condiciones que tienen que darse para buscar un bloque de M_p y llevarlo a una línea de M_{ca}.

Existen tres alternativas principales:

1) Por *demanda* ⇒ Se lleva un bloque a M_{ca} cuando se referencia desde la CPU alguna palabra de un bloque y éste no se encuentra en la M_{ca}.
2) *Anticipativa (Prebúsqueda - Prefetching)*:

    a. *Prebúsqueda siempre* ⇒ La primera vez que se referencia el bloque i se busca también el bloque i+1 (se produzca fallo o no).
    b. *Prebúsqueda por fallo* ⇒ Cuando se produce un fallo al acceder al bloque i se buscan los bloques i e i+1.

3) *Selectiva* ⇒ Consiste en etiquetar algún tipo de información para que nunca pueda ser enviada a la M_{ca} (información no cacheable). Por ejemplo: en un sistema multiprocesador con memoria compartida.

4. ALGORITMOS DE REEMPLAZAMIENTO

Cuando un nuevo bloque se transfiere a la memoria caché debe sustituir a uno de los ya existentes. En el caso de la *correspondencia directa*, la partición está determinada unívocamente y por lo tanto no se puede realizar ninguna elección. Por el contrario, en las técnicas de tipo asociativo es necesario un ***algoritmo de reemplazamiento***. Este mecanismo de sustitución debe realizarse totalmente por hardware y preferiblemente que la selección se haga toda ella durante el ciclo de memoria principal de búsqueda del nuevo bloque.

De forma general, los algoritmos de sustitución de bloques de la memoria caché se pueden clasificar en dos grandes grupos:

- Los que están basados en su utilización.
- Los que no toman en cuenta la utilización.

Además, un factor crítico es a menudo la cantidad de hardware que se necesita para implementar el algoritmo.

Entre los diferentes algoritmos que se han propuesto, merecen destacarse los cuatro siguientes:

a) *Bloque elegido de forma aleatoria* ⇒ Toma de forma aleatoria un bloque de la M_{ca} y lo sustituye por el bloque de la M_p que contiene a la palabra a la que se acaba de acceder ⇒ Ventaja: es muy fácil de implementar en hardware y es más rápido que la mayoría de los otros algoritmos ⇒ Desventaja: el bloque que es más probable que se vaya a utilizar otra vez tiene la misma probabilidad de ser sustituido por cualquier otro bloque (esta desventaja se ve mitigada cuando el tamaño de la memoria caché aumenta, ya que habría más bloques para ser sustituidos).
b) *Bloque utilizado menos frecuentemente (Algoritmo LFU [Least Frecuently Used)]* ⇒ Supone que los bloques que no se referencian con frecuencia no se necesitan tanto ⇒ Cada bloque de la memoria caché incorpora un contador que almacena el número de veces que dicho bloque se ha utilizado desde que fue transferido a la memoria caché; el bloque que tenga el mínimo valor es el que se sustituye. Cuando se llegue al valor máximo en uno de los contadores, para resolver este problema, se pueden dividir todos los valores de los contadores por dos (desplazamiento un lugar a la derecha), aunque se pierda el bit menos significativo (los valores pares e impares consecutivos quedarán

con el mismo valor) ⇒ Ventaja: el bloque que se utiliza frecuentemente es más probable que permanezca en la M_{ca} y no así el que no se referencia a menudo ⇒ Desventaja: penaliza en exceso a los bloques que se han transferido recientemente a la M_{ca} ya que necesariamente tendrán un contador con un valor muy bajo a pesar de que es probable que serán utilizados otra vez en el futuro inmediato; y, es más difícil de implementar en términos de hardware y es por lo tanto más costoso.

c) *Bloque más antiguo en memoria caché [Algoritmo FIFO (First In First Out)]* ⇒ Elimina de la M_{ca} el bloque que ha permanecido durante más tiempo ⇒ Se realiza de forma natural con una cola de direcciones de bloques; sin embargo, se puede implementar más fácilmente con contadores [como al principio, la memoria caché se va rellenando de forma ordenada, el primer bloque a caché se ubica en el bloque 0, el segundo en el 1, el tercero en el 2, … (de la caché o del conjunto, dependiendo de si es correspondencia totalmente asociativa o asociativa por conjuntos), el/los contador/es indicarán siempre el bloque próximo a sustituir (al comienzo, cuando se haya ocupado toda la caché o todo el conjunto, según correspondencia totalmente asociativa o asociativa por conjuntos, el contador, de la caché o del conjunto correspondiente, tendrá el valor cero; y, cada vez que se sustituya el bloque de la caché que indica el contador, éste se incrementará en una unidad para indicar el siguiente bloque a salir en la próxima sustitución]: en una memoria caché totalmente asociativa sólo es preciso un contador, y para una memoria caché asociativa por conjuntos se requiere un contador por cada conjunto. En ambos casos, los contadores deberán tener un número suficiente de bits para identificar completamente al bloque; el contador único en la correspondencia totalmente asociativa, debe tener capacidad para indicar cualquiera de los bloques que tenga la caché [de 0 a (C-1)]; cada uno de los q contadores asociados a cada conjunto en la correspondencia asociativa por conjuntos debe tener capacidad para indicar cualquiera de los bloques del conjunto [de 0 a (r-1)].

d) *Bloque utilizado menos recientemente [Algoritmo LRU (Least Recently Used)]* ⇒ Un bloque que no se ha utilizado durante largo periodo de tiempo tiene una probabilidad menor de ser utilizado en el futuro inmediato de acuerdo con el principio de localidad temporal. Este método retiene bloques en la M_{ca} que son más probables que puedan ser utilizados otra vez ⇒ Se emplea un mecanismo que almacena qué bloques han sido accedidos más recientemente. Una forma de implementar esta técnica es asociar un contador a cada bloque de la M_{ca} (cada vez que se accede a la M_{ca}, cada contador de bloque se incrementa en uno salvo el contador del bloque al que se accedió, que se inicializa a cero; de esta forma, el bloque cuyo contador tiene el máximo valor es el utilizado menos recientemente ⇒ El valor de cada contador indica la *edad de un bloque* desde la última vez que se referenció) ⇒ Ventaja: tiene el mejor rendimiento en relación con su coste cuando se le compara con las otras técnicas. Por ello es el que más se utiliza en los sistemas reales.

   El algoritmo para estos *registros de envejecimiento* se puede modificar para tomar en cuenta el hecho de que los contadores tienen un número fijo de bits y que sólo se precisa la edad relativa entre ellos. Si no se tuviese esto en cuenta se produciría un desbordamiento del contador que alteraría el resultado, convirtiendo al bloque que más tiempo hace que se utiliza en el que menos. El algoritmo es el siguiente:

   1) Cuando ocurre un acierto ⇒ El contador asociado con ese bloque se inicializa a cero (lo que indica que es el que se ha utilizado más recientemente); todos los contadores que tienen un valor más pequeño que

el que tenía el contador del bloque accedido se incrementan en uno; todos los contadores que tienen un valor mayor que el que tenía el contador del bloque accedido no se modifican.

2) Cuando ocurre un fallo y la M_{ca} no está llena ⇒ El contador asociado con el bloque que proviene de la M_p se inicializa a cero; todos los otros contadores se incrementan en uno.
3) Cuando ocurre un fallo y la M_{ca} está llena ⇒ Se sustituye el bloque que tiene el contador con el máximo valor y se inicializa a cero su contador; todos los otros contadores se incrementan en uno.

En la Tabla 2.1 se muestra un ejemplo de una memoria caché asociativa por conjuntos con 4 bloques/conjunto (r = 4). Un contador de 2 bits es suficiente para cada bloque. C_0, C_1, C_2 y C_3, representan los contadores en un conjunto. Al comienzo del algoritmo, todos los contadores se inicializan a cero. Se observa que cuando un bloque de la M_p sustituye a otro de la M_{ca} cambia el valor de la etiqueta que referencia de forma unívoca a cada bloque.

Nº acceso	Etiqueta	Acierto/Fallo	C_0	C_1	C_2	C_3	Acciones
			0	0	0	0	Inicialización
1º	5	Fallo	0	1	1	1	Bloque 0 rellenado
2º	6	Fallo	1	0	2	2	Bloque 1 rellenado
3º	5	Acierto	0	1	2	2	Bloque 0 accedido
4º	7	Fallo	1	2	0	3	Bloque 2 rellenado
5º	8	Fallo	2	3	1	0	Bloque 3 rellenado
6º	6	Acierto	3	0	2	1	Bloque 1 accedido
7º	9	Fallo	0	1	3	2	Bloque 0 sustituido
8º	10	Fallo	1	2	0	3	Bloque 2 sustituido

Tabla 2.1 Ejemplo de acceso a una memoria caché asociativa por conjuntos (4 bloques/conjunto). C_0, C_1, C_2 y C_3 representan los contadores en un conjunto.

5. ESTRATEGIA DE ESCRITURA

Para que pueda reemplazarse directamente un bloque en la memoria caché, es necesario saber primero si ha sido modificado en la M_{ca} y no en la M_p. Si no ha sido modificado, se puede reeemplazar directamente, se puede reescribir en él. En caso de haberse modificado (se ha realizado al menos una operación de escritura sobre una palabra de ese bloque de memoria caché), antes se debe actualizar el bloque correspondiente de la M_p. Hay una serie de estrategias de escritura, cada una de ellas con un coste y un rendimiento asociado. En general se plantean dos problemas:

a) Más de un dispositivo puede tener acceso a la memoria principal ⇒ Por ejemplo, un dispositivo con capacidad de acceso directo a memoria (DMAC). Si una palabra ha sido modificada solamente en la memoria caché, entonces la palabra de memoria principal correspondiente ya no es válida.
b) Varias CPU´s se conectan a un mismo bus y cada una de ellas tiene su propia memoria caché local. En este caso, si se cambia una palabra en una memoria caché, podría invalidar la misma palabra presente en otras memorias caché.

La estrategia de escritura o política de actualización determinan el instante en que se actualiza la información en M_p cuando se produce una escritura en M_{ca} ⇒ Problema de coherencia de la M_{ca} ⇒ Hay dos estrategias a seguir:

1. *Escritura inmediata (Write Through)* ⇒ Todas las operaciones de escritura se realizan tanto en la memoria principal como en la memoria caché, lo que asegura que sus contenidos son siempre válidos. Cualquier otro módulo CPU-

M_{ca} puede supervisar el tráfico con la memoria principal, para mantener la consistencia dentro de su propia memoria caché.

a. Ventajas:

- Realización muy sencilla.
- Asegura la consistencia (en sistemas monoprocesador).
- Supervisando el tráfico con la memoria principal, se puede mantener la coherencia entre las distintas cachés (en sistemas multiprocesadores).

b. Inconvenientes:

- Se genera mucho tráfico a memoria ⇒ Puede crear una saturación que disminuya significativamente el rendimiento del sistema.
- El procesador tiene que esperar (la diferencia entre los tiempos de escritura en M_{ca} y M_p) para que se complete la escritura ⇒ Una posible solución a este problema pasa por hacer uso de un buffer de escritura.

2. *Escritura aplazada o postescritura (Write Back)* ⇒ Se realizan las actualizaciones sólo en la memoria caché. Cuando se produce una modificación, se pone a 1 un **bit de actualización (dirty bit)** asociado con cada bloque de la memoria caché. Si se reemplaza un bloque se reescribe en la memoria principal si y sólo si el bit de actualización está a 1.

a. Ventajas:
- Produce menos tráfico de información en la M_p.
- Los accesos de escritura, en caso de acierto, se llevan a cabo a la velocidad de la M_{ca}.

b. Inconvenientes:

- El diseño es más complejo debido a que es necesario implementar el control del **bit de actualización**.
- Saber qué partes de la memoria principal y de otras cachés no son válidas.

Existen dos formas de actuar en el caso en que un ***acceso a escritura*** produzca un **fallo**:

a) *Escritura con ubicación (Write With Allocate)* ⇒ Se suele asociar con escritura aplazada ⇒ Consiste en llevar el bloque que produce el fallo, de M_p a M_{ca} y a continuación realizar la escritura en la M_{ca}.
b) *Escritura sin ubicación (Write With No Allocate)* ⇒ Se suele asociar con escritura inmediata ⇒ Consiste en realizar únicamente la escritura sobre la M_p cuando se produce un fallo.

6. TAMAÑO DEL BLOQUE

Otro parámetro importante en el diseño de la memoria caché es el *tamaño del bloque* ⇒ Tenemos dos efectos cuando aumentamos el tamaño del bloque (para una misma capacidad de caché, tenemos menos bloques):

1. ↑↑↑ *tamaño del bloque* ⇒ ↑↑↑ tasa de aciertos por *localidad espacial* (se incrementa la posibilidad de que sean accedidos, en el futuro inmediato, datos próximos a la palabra referenciada).
2. ↑↑↑ *tamaño del bloque* ⇒ ↓↓↓ tasa de aciertos por *localidad temporal* (hay palabras muy distantes, siendo menos probable que sean necesitadas a corto plazo) ⇒ ↑↑↑ utilización del *algoritmo de reemplazamiento de bloques*.

Debido a esos dos efectos contrarios, cuando se va aumentando el tamaño del bloque a partir de valores muy pequeños la tasa de aciertos inicialmente aumentará a causa del *principio de localidad espacial* ya que se incrementa la probabilidad de que sean accedidos en el futuro inmediato, datos próximos a la palabra referenciada. Sin embargo, a partir de un cierto tamaño del bloque, la tasa de aciertos comienza a disminuir, debido al *principio de localidad temporal* ya que la probabilidad de utilizar la información contenida en el bloque se hace menor que la probabilidad de reusar la información que se tiene que reemplazar (aumenta la utilización del algoritmo de reemplazamiento de bloques).

La relación entre tamaño de bloque y tasa de aciertos es compleja, y depende de las características de localidad de cada programa en particular. Por este motivo no se ha encontrado un valor óptimo definitivo, sin embargo, una elección razonable es entre 4 y 8 palabras.

7. NÚMERO DE CACHÉS

Cuando la memoria caché se integra en el mismo chip del procesador, puede resultar imposible aumentar su tamaño si el rendimiento del sistema no es el adecuado. En cualquier caso, ampliar el tamaño de la M_{ca} puede producir una caché que sea más lenta que la original. Una alternativa que se está utilizando cada vez más es introducir una segunda caché de mayor capacidad que se localiza entre la primera caché y la memoria principal. Esta *caché de segundo nivel* se denomina algunas veces *caché secundaria (caché off chip)*. Su tamaño suele ser de un orden de magnitud mayor que el de la *caché de primer nivel (Caché "on chip")*. En la Figura 2.13 se muestra el esquema de una memoria caché de dos niveles.

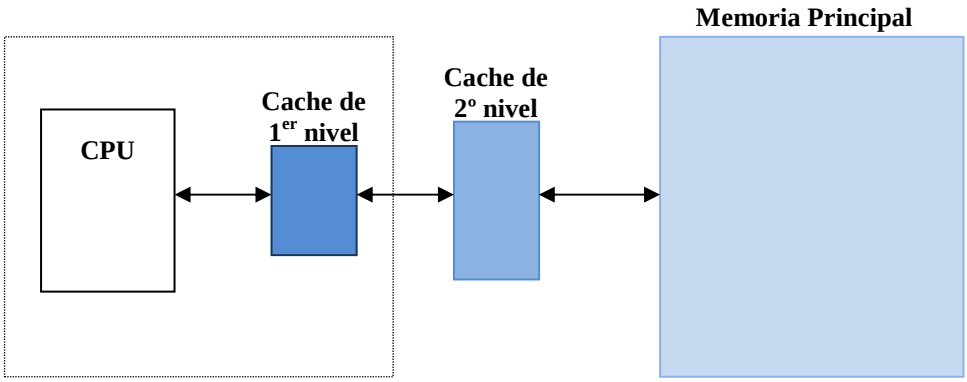


Figura 2.13 Memoria caché de dos niveles.

En una referencia a memoria, el procesador accederá a la caché de primer nivel. Si la información no se encuentra allí (se dice que ocurre un fallo en la M_{ca} de primer nivel), se accede a la caché de segundo nivel. Si tampoco la información solicitada se encuentra allí (se dice que ocurre un fallo en la M_{ca} de segundo nivel) habrá que acceder a la M_p. El bloque de memoria correspondiente se transfiere a la M_{ca} de segundo nivel y luego a la de primer nivel, de forma que existen, al menos inicialmente, dos copias del bloque de memoria $\Rightarrow$ Todos los bloques transferidos a la caché de primer nivel también existen en la caché de segundo nivel (*Principio de Inclusión*).

La política de reemplazamiento y la estrategia de escritura practicada en ambas cachés, normalmente, suelen ser el algoritmo LRU y la escritura inmediata para mantener las copias duplicadas.

La mayoría de las familias de microprocesadores proporcionan cachés de segundo nivel que emplean controladores independientes para las cachés externas al chip de la CPU. Así, el Intel 486 dispone del circuito controlador de caché 8291, y el Pentium del 82491. Motorota también posee controladores de cachés análogos.

En cuanto al número de memorias cachés en un sistema, también hay que diferenciar entre *cachés unificadas* y *cachés partidas*. En la *caché unificada*, los datos y las instrucciones se almacenan conjuntamente; en la *caché partida*, los datos se almacenan en una *caché de datos* y las instrucciones en una *caché de instrucciones*. En la Figura 2.14 se muestra un esquema de *caché partida*.

Las ventajas que encontramos en la *memoria caché partida* son:

- Se puede mejorar la tasa de aciertos puesto que las instrucciones presentan sobre todo *localidad espacial* y los datos *localidad temporal*.
- Permite acceso simultáneo a datos e instrucciones.

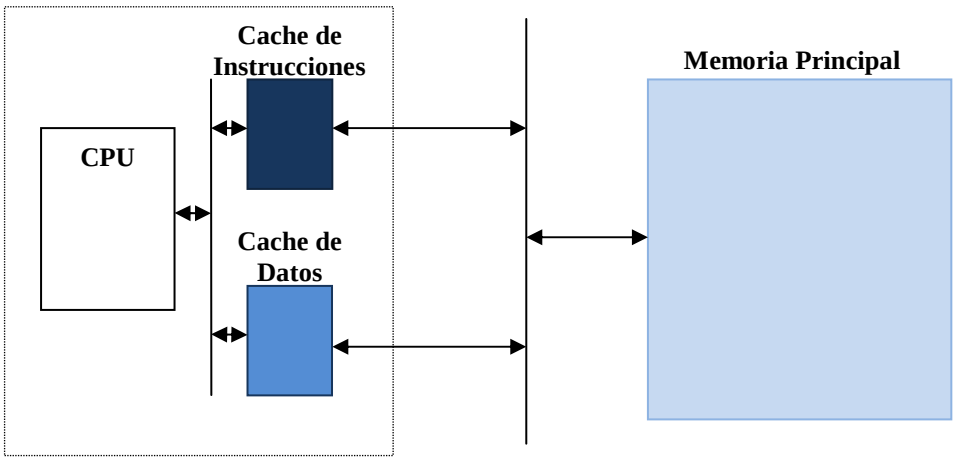


Figura 2.14 Memoria caché partida.

En los modernos sistemas microcomputadores nos podemos encontrar memorias cachés de dos niveles, siendo la de primer nivel partida. En la Figura 2.15 se muestra esquemáticamente el sistema de memoria de un sistema computador basado en el Intel Pentium.

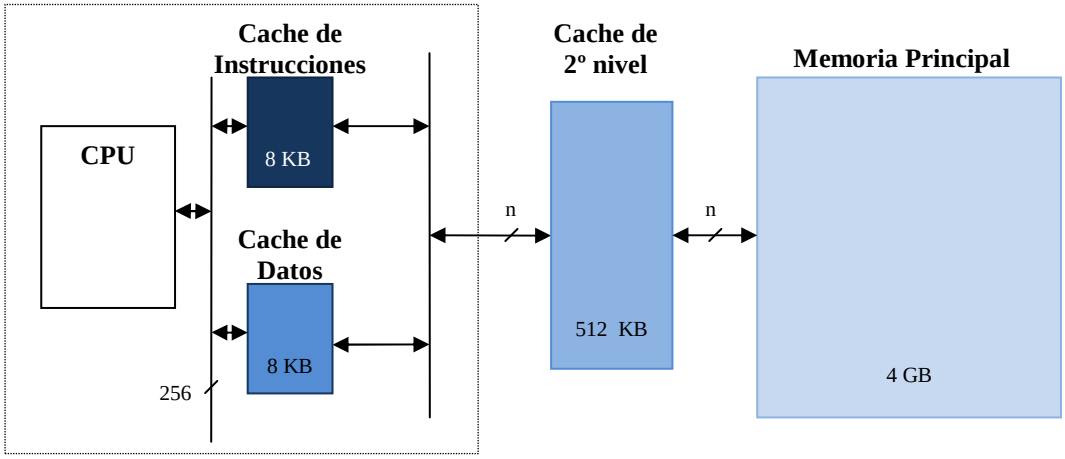


Figura 2.15 Sistema de memoria del Intel Pentium.

2.3 EVALUACIÓN DEL RENDIMIENTO DE UNA JERARQUÍA DE MEMORIA

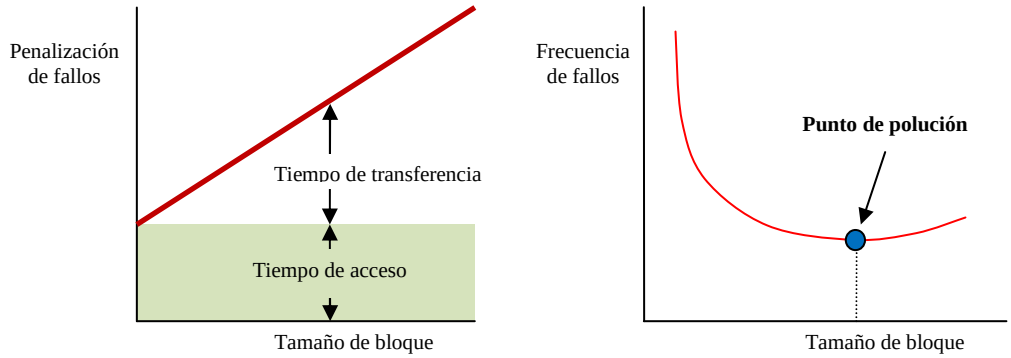
Debido a que el número de instrucciones es independiente del hardware (sin considerar la propia CPU, podría pensarse en evaluar el rendimiento de la CPU utilizando ese número. Sin embargo, las medidas indirectas del rendimiento han sido erróneamente utilizadas por muchos diseñadores de computadores. La tentación correspondiente de evaluar el rendimiento de la jerarquía de memoria es concentrarse en la ***frecuencia de fallos***, ya que ésta, también es independiente de la velocidad del hardware. La *frecuencia de fallos* puede ser tan engañosa como el *número de instrucciones* $\Rightarrow$ Una mejor medida del rendimiento de la jerarquía de memoria es el ***tiempo medio para acceder a memoria***:

Tiempo medio de acceso a memoria = Tiempo de acierto + Frecuencia de fallos · Penalización de fallo

Los componentes del *tiempo medio de acceso* se pueden medir bien en tiempo absoluto (por ejemplo en ns) o en número de ciclos de reloj.

El *tiempo medio de acceso a memoria* es todavía una medida indirecta del rendimiento; así, aunque sea una medida mejor que la *frecuencia de fallos*, no es un sustituto del *tiempo de ejecución*.

La relación entre tamaño de bloque y penalización de fallos, así como frecuencia de fallos se muestra abstractamente en la Figura 2.16. (Se supone que el tamaño de memoria del nivel superior no varía).



Penalización de fallos = Tiempo de acceso (a M.P) + Tiempo de transferencia

Punto de polución ≡ Punto donde la frecuencia de fallos comienza a aumentar con el incremento del tamaño de bloque. Las frecuencias de fallo disminuyen al incrementar el tamaño de bloque hasta que la parte no utilizada del bloque llega a ser tan grande que quita el sitio a información útil del nivel superior.

Figura 2.16 Tamaño de bloque frente a penalización de fallos y frecuencia de fallos.

El objetivo de una jerarquía de memoria es reducir el tiempo de ejecución, no de fallos. Por consiguiente, los diseñadores de computadores prefieren mejor un tamaño de bloque con tiempo de acceso medio menor, que una frecuencia de fallos más baja. En la Figura 2.17 se muestra la relación entre el tiempo medio de acceso a memoria y el tamaño de bloque.

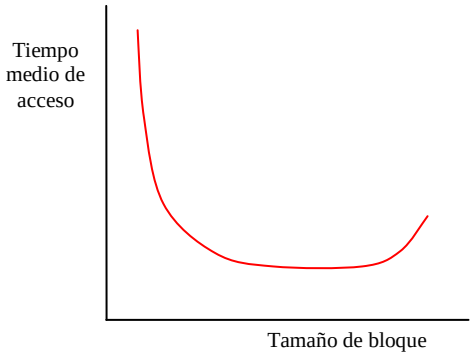


Figura 2.17 Relación entre tiempo medio de acceso a memoria y tamaño de bloque.

Por supuesto, el rendimiento global de la CPU es el último test de rendimiento, así que debe tenerse cuidado cuando se reduzca el tiempo medio de acceso a memoria para asegurarse que los cambios en la duración del ciclo de reloj y el CPI mejoran el rendimiento global así como el tiempo medio de acceso a memoria.

En la Tabla 2.2 se muestran los valores típicos de los parámetros que definen la jerarquía de memoria caché en estaciones de trabajo y minicomputadores.

Parámetros	Rango de valores
Tamaño de bloque (línea)	4 – 128 bytes
Tiempo de acierto	1 – 4 ciclos de reloj (normalmente 1)
Penalización de fallo ▪ Tiempo de acceso ▪ Tiempo de transferencia	8 – 32 ciclos de reloj ▪ 6 – 10 ciclos de reloj ▪ 2 – 22 ciclos de reloj
Frecuencia de fallo	1 % - 20 %
Tamaño de caché	1 KB – 256 KB

Tabla 2.2 Valores típicos de parámetros clave de la jerarquía de memoria caché en estaciones de trabajo y minicomputadores de 1990.

LAS FUENTES DE FALLOS DE LA CACHÉ

Un modelo intuitivo de comportamiento de la caché atribuye todos los fallos a una de tres fuentes:

- ***Forzosos*** *(Compulsory)* ⇒ El primer acceso a un bloque no está en la caché; así que debe ser traído a la caché ⇒ *Fallos de arranque en frío* o *fallos de primera referencia*.
- ***Capacidad*** *(Capacity)* ⇒ La caché no puede contener todos los bloques necesarios durante la ejecución de un programa.
- ***Conflicto*** *Conflict)* ⇒ Si la estrategia de ubicación de bloques es asociativa por conjuntos o de correspondencia directa, los fallos de conflicto (además de los forzosos y de capacidad), ocurrirán, ya que se puede descartar un bloque y posteriormente recuperarlo si a un conjunto le corresponden demasiados bloques ⇒ *Fallos de colisión*.

En la Tabla 2.3 se muestra la frecuencia total de fallos para cada tamaño de caché y porcentaje de cada fallo de acuerdo con las <<tres C>>.

La Figura 2.18 presenta los mismos datos gráficamente. El gráfico izquierdo muestra frecuencias absolutas de fallos y el derecho dibuja el porcentaje de todos los fallos por tamaño de caché.

Una vez identificados los tres tipos de fallos, nos hacemos la siguiente pregunta: ¿qué puede hacer un diseñador de computadores con respecto a ellos? ⇒ Conceptualmente, los *fallos de conflicto* son los más fáciles: la ubicación completamente asociativa evita todos los *fallos de conflicto*. Sin embargo, la asociatividad es cara en hardware y puede ralentizar el tiempo de acceso (ver ejemplo anterior), llevando a una disminución del rendimiento global ⇒ Hay poco que hacer sobre los *fallos de capacidad* excepto comprar mayores chips de memoria. Si la memoria de nivel superior es mucho más pequeña de la que se necesita para un programa, y se emplea un porcentaje significativo del tiempo en transferir datos entre los dos niveles de la jerarquía, la jerarquía de memoria se dice ***castigada*** *(thrash)*; la máquina corre próxima a la velocidad de la memoria de nivel más bajo, o quizás aún más lenta debido a la sobrecarga de fallos ⇒ Hacer bloques grandes reduce el número de *fallos forzosos*, pero puede incrementar los *fallos de conflicto*.

Tamaño cache	Grado asociatividad	Frecuencia total fallos	Componentes de la frecuencia de fallos (porcentaje relativo)					
			Forzoso		Capacidad		Conflicto	
1 KB	1 vía	0.191	0.009	5%	0.141	73%	0.042	22%
1 KB	2 vías	0.161	0.009	6%	0.141	87%	0.012	7%
1 KB	4 vías	0.152	0.009	6%	0.141	92%	0.003	2%
1 KB	8 vías	0.149	0.009	6%	0.141	94%	0.000	0%
2 KB	1 vía	0.148	0.009	6%	0.103	70%	0.036	24%
2 KB	2 vías	0.122	0.009	7%	0.103	84%	0.010	8%
2 KB	4 vías	0.115	0.009	8%	0.103	90%	0.003	2%
2 KB	8 vías	0.113	0.009	8%	0.103	91%	0.001	1%
4 KB	1 vía	0.109	0.009	8%	0.073	67%	0.027	25%
4 KB	2 vías	0.095	0.009	9%	0.073	77%	0.013	14%
4 KB	4 vías	0.087	0.009	10%	0.073	84%	0.005	6%
4 KB	8 vías	0.084	0.009	11%	0.073	87%	0.002	3%
8 KB	1 vía	0.087	0.009	10%	0.052	60%	0.026	30%
8 KB	2 vías	0.069	0.009	13%	0.052	75%	0.008	12%
8 KB	4 vías	0.065	0.009	14%	0.052	80%	0.004	6%
8 KB	8 vías	0.063	0.009	14%	0.052	83%	0.002	3%
16 KB	1 vía	0.066	0.009	14%	0.038	57%	0.019	29%
16 KB	2 vías	0.054	0.009	17%	0.038	70%	0.007	13%
16 KB	4 vías	0.049	0.009	18%	0.038	76%	0.003	6%
16 KB	8 vías	0.048	0.009	19%	0.038	78%	0.001	3%
32 KB	1 vía	0.050	0.009	18%	0.028	55%	0.013	27%
32 KB	2 vías	0.041	0.009	22%	0.028	68%	0.004	11%
32 KB	4 vías	0.038	0.009	23%	0.028	73%	0.001	4%
32 KB	8 vías	0.038	0.009	24%	0.028	74%	0.001	2%
64 KB	1 vía	0.039	0.009	23%	0.019	50%	0.011	27%
64 KB	2 vías	0.030	0.009	30%	0.019	65%	0.002	5%
64 KB	4 vías	0.028	0.009	32%	0.019	68%	0.000	0%
64 KB	8 vías	0.028	0.009	32%	0.019	68%	0.000	0%
128 KB	1 vía	0.026	0.009	34%	0.004	16%	0.013	50%
128 KB	2 vías	0.020	0.009	46%	0.004	21%	0.006	33%
128 KB	4 vías	0.016	0.009	55%	0.004	25%	0.003	20%
128 KB	8 vías	0.015	0.009	59%	0.004	27%	0.002	14%

Tabla 2.3 Frecuencia total de fallos para cada tamaño de cache y porcentaje de cada fallo de acuerdo con las <<tres C>> [Compulsory (*forzoso*), Capacity (*capacidad*), Conflict (*conflicto*). Los *fallos forzosos* son independientes del tamaño de la cache, mientras que los *fallos de capacidad* decrecen cuando la capacidad aumenta. Hill (1987) midió esta traza utilizando bloques de 32 bytes y reemplazo LRU. Observar que la regla empírica de cache 2:1 está soportada por las estadísticas de esta tabla: una cache de correspondencia directa de tamaño N tiene aproximadamente la misma frecuencia de fallos que una cache asociativa por conjuntos de 2 vías de tamaño N/2.

Las <<tres C>> dan una visión de las causas de los fallos, pero este modelo simple tiene sus límites:

- Un fallo puede ir de una categoría a otra cuando cambian los parámetros.

Ignoran la política de reemplazo, ya que es difícil de modelar y, en general, es de menor significado. En circunstancias específicas, la política de reemplazo realmente puede llevar a comportamientos anómalos, como frecuencias más pobres de fallos para mayor asociatividad, lo que es directamente contradictorio con el modelo de las <<tres C>>.

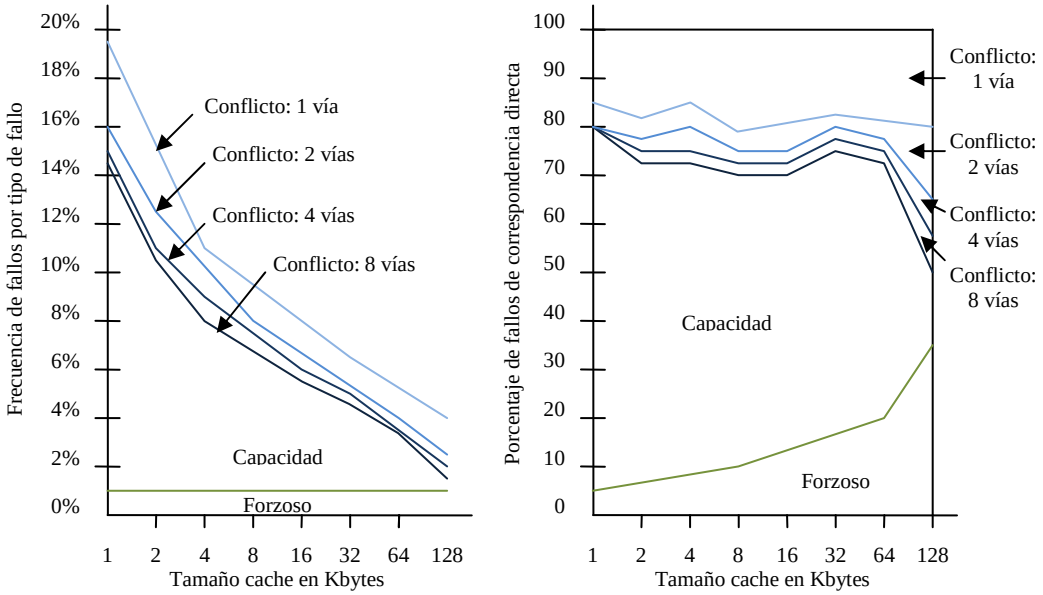


Figura 2.18. Frecuencia total de fallos (izquierda) y distribución de la frecuencia de fallos (derecha) para cada tamaño de cache de acuerdo con las tres C para los datos de la figura 3.18. El diagrama izquierdo es la frecuencia real de fallos, mientras que el derecho está escalado respecto a la frecuencia de fallos de correspondencia directa.

ELECCIONES PARA TAMAÑOS DE BLOQUE EN CACHÉS

Las Figuras 2.16 y 2.17 muestran en abstracto la relación del tamaño de bloque con la frecuencia de fallos y tiempos de acceso a memoria ⇒ Tamaños de bloques mayores reducen los *fallos forzosos*, como sugiere el principio de localidad espacial. Al mismo tiempo, bloques mayores también reducen el número de bloques de la caché, incrementando los *fallos de conflicto*.

CACHÉ DE SÓLO INSTRUCCIONES Y CACHÉ DE SÓLO DATOS FRENTE A CACHÉ UNIFICADA

La CPU sabe si está emitiendo la dirección de una instrucción o de un dato; así, puede haber puertos separados para ambos, casi doblando el ancho de banda entre la caché y la CPU. Las cachés separadas además ofrecen la oportunidad de optimizar cada caché separadamente: diferentes capacidades, tamaños de bloque y asociatividades pueden conducir a un mejor rendimiento ⇒ Dividir entonces afecta al coste y rendimiento más allá de lo que está indicado por el cambio de las frecuencias de fallos (la frecuencia de fallos para instrucciones difiere de la frecuencia de fallos para datos).

La Tabla 2.4 muestra cómo las cachés de instrucciones tienen menor frecuencia de fallos que las cachés de datos ⇒ La separación de instrucciones y datos elimina fallos debidos a conflictos entre los bloques de instrucciones y los bloques de datos, pero, al dividir, también se fija el espacio de caché dedicado a cada tipo ⇒ Para hacer una buena comparación, la suma de las capacidades de las cachés de instrucciones y de datos separadas debe ser igual al tamaño de la caché unificada con la que se compara ⇒ Para calcular la frecuencia media de fallos en cachés separadas de instrucciones y datos se necesita conocer el porcentaje de referencia a memoria en cada caché.

Tamaño	Sólo instrucciones	Sólo datos	Unificada
0.25 KB	22.2%	26.8%	28.6%
0.50 KB	17.9%	20.9%	23.9%
1 KB	14.3%	16.0%	19.0%
2 KB	11.6%	11.8%	14.9%
4 KB	8.6%	8.7%	11.2%
8 KB	5.8%	6.8%	8.3%
16 KB	3.6%	5.3%	5.9%
32 KB	2.2%	4.0%	4.3%
64 KB	1.4%	2.8%	2.9%
128 KB	1.0%	2.1%	1.9%
256 KB	0.9%	1.9%	1.6%

Tabla 2.4 Frecuencia de fallos para caches de distintos tamaños de sólo datos, sólo instrucciones, y unificadas. Los datos son para una cache asociativa de 2 vías utilizando reemplazo LRU con bloques de 16 bytes para un promedio de trazas de usuario/sistema en el VAX-11 y trazas de sistema en el IBM 370 (Hill 1987). El porcentaje de referencia a instrucciones en estas trazas es aproximadamente del 53%.

Ejemplo para comparación de cachés separadas de instrucciones y datos con caché unificada. ¿Cuál tiene la frecuencia de fallos más baja: una caché de instrucciones de 16 KB con una caché de datos de 16 KB o una caché unificada de 32 KB?. Suponer que el 53% de las referencias son instrucciones.

Solución: Haciendo uso de la tabla 2.4, la frecuencia global de fallos para las cachés divididas es:

$$53\% \cdot 3.6\% + (100 - 53)\% \cdot 5.3\% = 4.399\%$$

y, según la misma tabla, una caché unificada de 32 KB tiene una frecuencia de fallos ligeramente más baja, de 4.3%.

2.4 MEMORIA PRINCIPAL

Suponiendo que solo haya un nivel de caché, la *memoria principal* es el siguiente nivel en la jerarquía. La *memoria principal* satisface las demandas de las cachés y unidades vectoriales, y sirve como interfaz de E/S, ya que es el destino de la entrada, así como la fuente para la salida.

De forma distinta a las cachés, las medidas de rendimiento de la *memoria principal* hacen énfasis en la ***latencia*** y ***ancho de banda***:

- La ***latencia*** de la *memoria principal* (que afecta a la penalización de fallos de la caché) es la preocupación primordial de la caché. Cuando los bloques de la caché crecen desde 4-8 bytes a 64-256 bytes, el ***ancho de banda*** de la *memoria principal* también llega a ser importante para las cachés.
- El ***ancho de banda*** de la *memoria principal* es la preocupación primordial de la E/S y de las unidades vectoriales.

La ***latencia de la memoria*** se expresa utilizando dos medidas:

- ***Tiempo de acceso*** $\Rightarrow$ Es el tiempo desde que se pide una lectura hasta que llega la palabra deseada.
- ***Duración del ciclo*** $\Rightarrow$ Es el tiempo mínimo entre peticiones a memoria.

Actualmente, la *memoria principal* de la mayor parte de los computadores se construye con dispositivos de memoria DRAM (RAM dinámica) [tienen señales $\overline{\text{RAS}}$ y $\overline{\text{CAS}}$; necesita operación de refresco (cada 2 ms, por ejemplo) → el sistema de memoria no está disponible porque está enviando una señal para refrescar cada chip]. Prácticamente todas las cachés utilizan dispositivos de memoria SRAM (RAM estática).

Para memorias diseñadas con tecnologías comparables, la capacidad de las DRAM es aproximadamente 16 veces la de la SRAM, y el tiempo del ciclo de las SRAM es de 8 a 16 veces más rápido que las DRAM.

Amdahl sugirió la regla empírica que la capacidad de memoria debería crecer linealmente con la velocidad de la CPU para mantener un sistema equilibrado. Desgraciadamente, el rendimiento de las DRAM ha crecido a una velocidad mucho más lenta. La diferencia de rendimiento de CPU-DRAM es claramente un problema; no se puede ignorar una parte de la computación mientras tratamos de acelerar el resto → simplemente hacer cachés mayores no puede eliminar el problema. Se pueden tomar las siguientes medidas:

- Con la ***organización de la memoria caché***: *ubicación de subbloques*, *buffers de escritura*, *búsqueda fuera de orden*, *cachés direccionadas virtualmente*, *cachés de dos niveles* y *aspectos relativos a la coherencia de la caché*. (No se estudiarán por falta de tiempo).
- Con la ***organización de la memoria principal***: *aumentar el ancho de banda* y *entrelazado de la memoria*. (Se estudian a continuación).

ORGANIZACIONES PARA MEJORAR EL RENDIMIENTO DE LA MEMORIA PRINCIPAL

- **Memoria principal más ancha** ⇒ Las cachés están con frecuencia organizadas con una anchura de una palabra porque la mayoría de los accesos de la CPU son de ese tamaño. A su vez, la memoria principal tiene el ancho de una palabra para que coincida con la anchura de la caché ⇒ Duplicar o cuadruplicar el ancho de memoria, por tanto, duplicará o cuadriplicará el ancho de banda de memoria (Figura 2.19).

  Inconvenientes:

  1. Hay coste en el bus más ancho. La CPU accederá todavía a la caché una palabra cada vez, por ello, se necesita ahora un multiplexor entre la caché y la CPU (y ese multiplexor puede estar en el camino crítico de temporización). También se puede colocar el multiplexor entre la caché y el bus si la caché es más rápida que el bus.
  2. Inconveniente al expandir la memoria el usuario ⇒ El incremento mínimo se duplica o cuadruplica.
  3. Dificultades en la corrección de errores ⇒ Las memorias con corrección de errores tienen dificultades con las escrituras en una parte del bloque protegido (por ejemplo, una escritura de un byte); el resto de los datos se debe leer para que el nuevo código de corrección de errores se pueda calcular y almacenar cuando se escriba el dato. Si la corrección de errores se hace sobre la anchura completa, la memoria más ancha incrementará la frecuencia de estas secuencias de <<leer-modificar-escribir>>, porque más escrituras se convierten en escrituras parciales

de bloques ⇒ Muchos diseños de memoria separan la corrección de errores a tamaños correspondientes al de las escrituras.

- **Memoria entrelazada** ⇒ Los chips de memoria se pueden organizar en bancos para leer o escribir múltiples palabras a la vez, en lugar de una sola palabra. Los bancos son de una palabra de ancho para que la anchura del bus y de la caché no necesiten cambiar, pero enviando direcciones a varios bancos, les permite a todos leer simultáneamente ⇒ Los bancos también son útiles en las escrituras. Aunque las escrituras muy seguidas normalmente tendrán que esperar a que acaben las escrituras anteriores, los bancos permiten un ciclo de reloj para cada escritura, siempre que no estén destinadas al mismo banco ⇒ La correspondencia entre direcciones y bancos afecta el comportamiento del sistema de memoria. Esta correspondencia se denomina *factor de entrelazado*. Esto optimiza los accesos secuenciales a memoria ⇒ Un fallo de lectura de caché es un caso ideal para una lectura entrelazada a nivel de palabra, ya que las palabras de un bloque se leen secuencialmente. Las cachés de postescritura hacen escrituras así como lecturas secuenciales, obteniendo incluso más eficiencia de la memoria entrelazada ⇒ La motivación original para los bancos de memoria fue entrelazar accesos secuenciales. Una razón adicional es permitir múltiples accesos independientes. Controladores múltiples de memoria permiten bancos (o conjuntos de bancos de entrelazados por palabras) operar independientemente (por ejemplo, un dispositivo de entrada puede utilizar un controlador y su memoria caché puede utilizar otro, y una unidad vectorial puede utilizar un tercero).

  Inconvenientes:

  1. Cuando aumenta la capacidad por chip de memoria, hay menos chips en un sistema de memoria del mismo tamaño, haciendo mucho más caros múltiples bancos.
  2. La dificultad de la expansión de la memoria principal ⇒ Como el hardware de control de memoria probablemente necesitará bancos de igual tamaño, duplicar la memoria principal probablemente será el incremento mínimo.

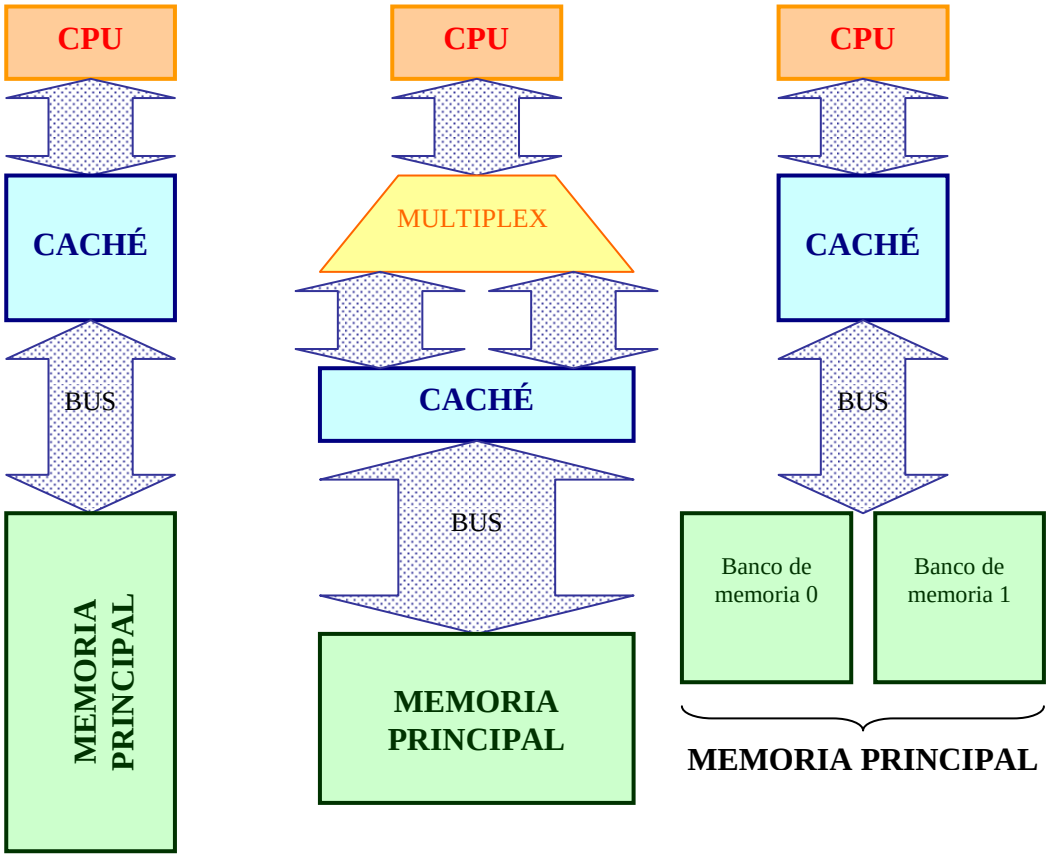
Ejemplo que ilustra cómo se satisface un fallo de caché en las distintas organizaciones. Suponer que el rendimiento de la organización básica de memoria es:

1 ciclo de reloj para enviar la dirección
6 ciclos de reloj para el tiempo de acceso por palabra
1 ciclo de reloj para enviar una palabra de datos
Bloque de caché de 4 palabras
Palabra de memoria de 32 bits

Para la *organización de memoria de una palabra de ancho*, la penalización de fallos es de: (1+6+1) ciclos/1 palabra · 4 palabras/bloque = **32 ciclos/bloque**.

Para la *organización de memoria ancha* (con una anchura de memoria principal de 2 palabras), la penalización de fallos en nuestro ejemplo se reduciría, cayendo a la mitad:(1+6+1) ciclos/2 palabras · 4 palabras/bloque = **16 ciclos/bloque**.

Para la *organización de memoria entrelazada* (con cuatro bancos), enviando una dirección a los cuatro bancos (a la vez), la penalización de fallos quedaría aún menor: (1+6+1·4) ciclos/bloque = **11 ciclos/bloque**.



a) Organización de memoria de una palabra de ancho

b) Organización de memoria ancha

c) Organización de memoria entrelazada

Figura 2.19 Tres ejemplos de anchura de bus, ancho de memoria, y entrelazado de memoria para lograr mayor ancho de banda de memoria. a) es el diseño más simple, siendo todo de la anchura de una palabra; b) muestra una memoria, bus y caché más anchos; mientras que c) muestra una caché y bus delgados con una memoria entrelazada.

ENTRELAZADO DE DRAM ESPECÍFICO PARA MEJORAR EL RENDIMIENTO DE LA MEMORIA PRINCIPAL

Los tiempos de acceso a las DRAM se dividen en accesos a filas y accesos a columnas. Las DRAM disponen de un buffer de una fila de bits dentro de la DRAM para los accesos a las columnas. Esta fila es habitualmente la raíz cuadrada del tamaño de la DRAM ⇒ Todas las DRAM vienen con señales de temporización opcional que permiten accesos repetidos al buffer sin un tiempo de acceso a filas. Hay tres versiones para esta optimización:

1. *Modo nibble* ⇒ La DRAM puede proporcionar tres bits extras de posiciones secuenciales para cada acceso a filas.
2. *Modo de página* ⇒ El buffer actúa como una SRAM; cambiando las direcciones de las columnas, se puede acceder a bits aleatorios en el buffer hasta el siguiente acceso a fila o tiempo de refresco.
3. *Columna estática* ⇒ Muy similar al modo de página, excepto que no es necesario acertar la línea de strobe de acceso a columnas cada vez que cambie la dirección de la columna; esta opción se ha denominado SCRAM, para la DRAM de columna estática.

A partir de la DRAM de 1 Mbit, la mayoría de los dispositivos pueden realizar alguna de las tres opciones anteriores, seleccionando la optimización en el momento del encapsulado. Estas operaciones cambian la definición del tiempo del ciclo para la DRAM. La Tabla 2.5 muestra la duración tradicional del ciclo más la velocidad más rápida entre accesos en el modo optimizado.

	Acceso a fila				
Tamaño del chip	**DRAM más lenta**	**DRAM más rápida**	**Acceso a columna**	**Tiempo de ciclo**	**Tiempo nibble, página, columna estática (optimizada)**
64 Kbits	180 ns	150 ns	75 ns	250 ns	150 ns
256 Kbits	150 ns	120 ns	50 ns	220 ns	100 ns
1 Mbits	120 ns	100 ns	25 ns	190 ns	50 ns
4 Mbits	100 ns	80 ns	20 ns	165 ns	40 ns
16 Mbits	≈ 85 ns	≈ 65 ns	≈ 15 ns	≈ 140 ns	≈ 30 ns

Tabla 2.5 Duración del ciclo de DRAM para los accesos optimizados. A partir de la DRAM de 1 Mbit, el tiempo de ciclo optimizado es aproximadamente cuatro veces más rápido que el tiempo de ciclo no optimizado. Es tan rápido que el modo de página fue renombrado como *modo rápido de página*. El tiempo de ciclo optimizado es el mismo sin importar cuál de los tres modos optimizados se utilice.

La ventaja de estas optimizaciones es que utilizan la circuitería incluida en las DRAM, añadiendo poco coste al sistema, mientras logran casi una mejora de cuatro veces en el ancho de banda.

De forma distinta a las memorias entrelazadas tradicionalmente, no hay desventajas al utilizar estos modos de DRAM cuando éstas aumentan en capacidad, ni existe el problema del mínimo incremento de expansión en memoria principal.

Otra posibilidad de mejorar el rendimiento de las memorias DRAM consiste en emplear dispositivos de memoria DRAM que no multiplexan las líneas de dirección.

2.5 MEJORA DEL RENDIMIENTO DE LA MEMORIA CACHÉ

La creciente separación entre las velocidades de la CPU y la memoria principal han atraído la atención de muchos arquitectos de computadores. Después de tomar algunas decisiones, fáciles al principio, el arquitecto de computador se enfrenta a un dilema triple cuando intenta reducir el tiempo medio de acceso:

1. Incrementar el tamaño de bloque, no mejora el tiempo medio de acceso; ya que, la menor frecuencia de fallos no compensa la mayor penalización de los fallos.
2. Hacer la memoria caché mayor, la haría más lenta; ya que pone en peligro la velocidad de reloj de la CPU.
3. Hacer la memoria caché más asociativa también la haría más lenta; ya que pone en peligro de nuevo la velocidad de reloj de la CPU.

Existe un amplio conjunto de técnicas para mejorar el rendimiento de la memoria caché:

1. Ubicación de subbloques
2. Buffers de escritura

3. Búsqueda fuera de orden
4. Cachés direccionadas virtualmente
5. Cachés de dos niveles
6. Reducción de limpiezas de caché
7. Reducción del tráfico del bus

CACHÉS DE DOS NIVELES. **Reducción de la penalización de fallos**.

Las CPUs cada vez son más rápidas y las memorias principales cada vez son mayores y, estas memorias principales cada vez son más lentas con respecto a las CPUs cada vez más rápidas. El arquitecto de computadores se plantea qué hacer con la memoria caché:

- Caché más rápida para mantenerse a la altura de la velocidad de la CPU.

o

- Caché de más capacidad para superar el creciente desnivel entre la CPU y la memoria principal.

La respuesta es: ambas cosas, añadiendo otro nivel de caché. Con dos niveles de caché se consigue:

1. Que la caché de primer nivel (la más cercana a la CPU) pueda tener coincidencia de duración de ciclo de reloj con la CPU, haciéndola más pequeña que una caché de un nivel.
2. Que la caché de segundo nivel pueda capturar muchos accesos que irían a la memoria principal, haciéndola bastante más grande que una caché de un nivel.

Las definiciones para un segundo nivel de caché no son siempre sencillas:

- *Tiempo medio de acceso a memoria*.

$$Tiempo\ medio\ acceso\ memoria =$$
$$= Tiempo\ acierto_{L1} + Frecuencia\ \ fallos_{L1} \cdot Penalización\ \ fallos_{L1}$$

$$Penalización\ \ fallos_{L1} =$$
$$= Tiempo\ acierto_{L2} + Frecuencia\ \ fallos_{L2} \cdot Penalización\ \ fallos_{L2}$$

$$Tiempo\ medio\ acceso\ memoria = Tiempo\ acierto_{L1} +$$
$$Frecuencia\ \ fallos_{L1} \cdot \left(Tiempo\ acierto_{L2} + Frecuencia\ \ fallos_{L2} \cdot Penalización\ \ fallos_{L2}\right)$$

- *Frecuencia local de fallos*. El número de fallos de la caché dividido por el número total de accesos a esta caché. (La *frecuencia fallos*$_{L2}$)
- *Frecuencia global de fallos*. El número de fallos de la caché dividido por el número total de accesos a memoria generados por la CPU. (La *frecuencia fallos*$_{L1}$ · *frecuencia fallos*$_{L2}$)

Ejemplo que muestra el cálculo de la frecuencia local de fallos y la frecuencia global de fallos. Suponer que en 1000 referencias a memoria hay 40 fallos en la caché de primer nivel y 20 fallos en la de segundo nivel. ¿Cuáles son las distintas frecuencias de fallos?

Solución: La frecuencia de fallos para la caché de primer nivel es:

$$Frecuencia\ \ fallos_{L1} = \frac{40\ \ fallos}{1000\ referencias\ memoria} = 4\ \%$$

La frecuencia local de fallos para la caché de segundo nivel es:

$$Frecuencia\ local\ \ fallos_{L2} = \frac{n^{o}\ \ fallos\ caché_{L2}}{n^{o}\ total\ accesos_{L2}} = \frac{20\ \ fallos}{40\ accesos_{L2}} = 50\ \%$$

La frecuencia global de fallos de la caché de segundo nivel es:

$$Frecuencia\ global\ \ fallos_{L2} = \frac{20\ \ fallos}{1000\ referencias\ memoria} = 2\ \% =$$

$$= Frecuencia\ \ fallos_{L1} \cdot Frecuencia\ \ fallos_{L2} = 4\ \% \cdot 50\ \% = 2\ \%$$

Las Figuras 2.20 y 2.21 muestran cómo las frecuencias de fallos y el tiempo de ejecución relativo cambian con el tamaño de la caché de segundo nivel.

La Tabla 2.6 muestra los parámetros típicos de las cachés de segundo nivel.

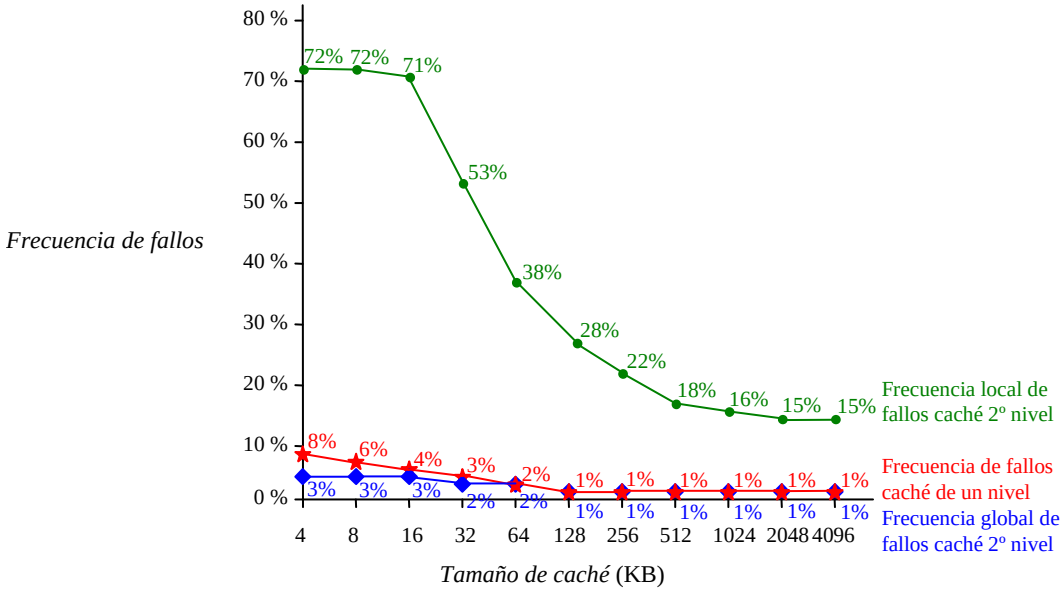


Figura 2.20 Frecuencia de fallos frente a tamaño de caché. La gráfica muestra la frecuencia de fallos de una caché de un solo nivel junto con la frecuencia local y frecuencia global de fallos de una caché de segundo nivel utilizando una caché de primer nivel de 32 KB. Las cachés de segundo nivel menores de los 32 KB del primer nivel tienen alta frecuencia de fallos. A partir de los 256 KB la caché única y las frecuencias globales de fallos de la caché de segundo nivel son virtualmente idénticas. (Datos recogidos por Przybylski [1990] utilizando trazas de programas de sistema y usuario del VAX y del MIPS R2000, del libro "*Arquitectura de Computadores. Un enfoque cuantitativo*", de Hennessy y Patterson

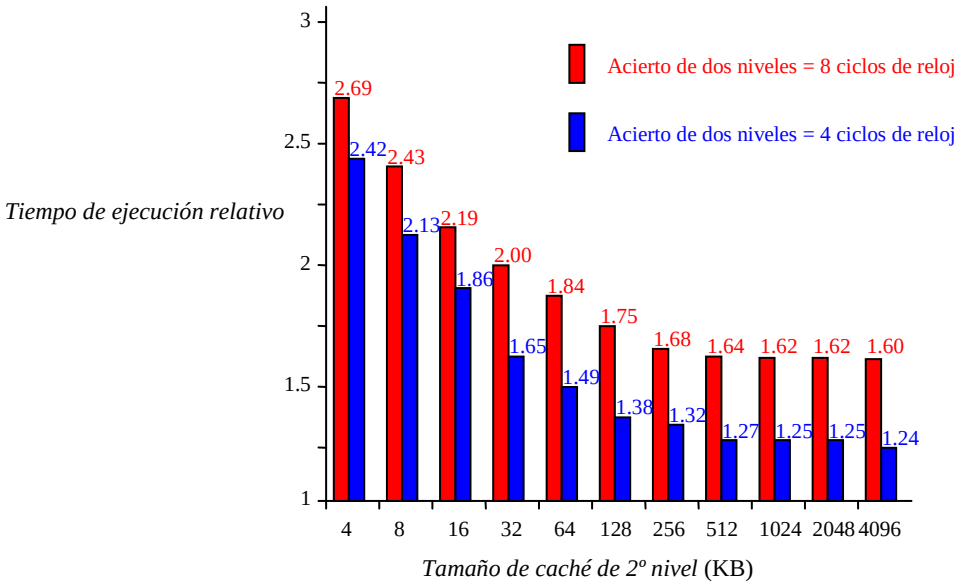


Figura 2.21 Tiempo de ejecución relativo según el tamaño de la caché de segundo nivel. Przybylski recopiló los datos utilizando una caché de postescritura, de 32 KB de caché de primer nivel y variando el tamaño de la caché de segundo nivel. Las dos barras son para diferentes tiempos de acierto en la caché de segundo nivel. El tiempo de ejecución de referencia de 1 es para una caché de segundo nivel de 4096 KB con una latencia de 1 ciclo de reloj en un acierto de segundo nivel. Empleó trazas de programas de sistema y usuario del VAX y del MIPS R2000, del libro "*Arquitectura de Computadores. Un enfoque cuantitativo*", de Hennessy y Patterson.

Parámetro	**Valor típico**
Tamaño de bloque (línea)	32 – 256 bytes
Tiempo de acierto	4 – 10 ciclos de reloj
Penalización de fallo: ▪ Tiempo de acceso ▪ Tiempo de transferencia	30 – 80 ciclos de reloj ▪ 14 – 18 ciclos de reloj ▪ 16 – 62 ciclos de reloj
Frecuencia local de fallos	15 – 30 %
Tamaño de caché	256 KB – 4 MB

Tabla 2.6 Valores típicos de los parámetros de las cachés de segundo nivel en una jerarquía de memoria.

Con las definiciones dadas para las cachés de segundo nivel, se pueden considerar sus parámetros. La diferencia principal entre los dos niveles de caché es que, la velocidad de la caché de primer nivel afecta a la frecuencia de reloj de la CPU, mientras que la velocidad de la caché de segundo nivel sólo afecta a la penalización de fallos de la caché de primer nivel. Por tanto, pueden considerarse muchas alternativas en la caché del segundo nivel que podrían no ser adecuadas para la caché de primer nivel. Pero, hay que hacerse la siguiente pregunta para la caché del segundo nivel: ¿Disminuirá la parte del CPI correspondiente al tiempo medio de acceso a memoria?

El parámetro a tener en cuenta en primer lugar para las cachés de segundo nivel es el *tamaño*. Como todo lo de la caché de primer nivel está probablemente en la caché de segundo nivel, ésta debería ser mayor que la de primer nivel. Si las cachés de segundo nivel son sólo un poco mayores, la frecuencia local de fallos de esta caché de segundo nivel será alta. Por tanto, se podría concluir que la caché de segundo nivel que nos interesa debe ser de tamaño muy grande; pero, si la caché de segundo nivel es muchísimo mayor que la de primer nivel, entonces la frecuencia global de fallos de la de segundo nivel es aproximadamente la misma que en una caché mononivel del

mismo tamaño (ver Figura 2.24). Es decir, aunque la caché de segundo nivel debe ser más grande que la de primer nivel, no puede llegar a ser tan excesivamente grande como para no interesar una caché de dos niveles.

En segundo lugar, cuando la caché de segundo nivel tiene un gran tamaño, prácticamente no se tienen fallos de capacidad, prestando atención a los fallos forzosos y a algunos fallos de conflicto; nos preguntamos si la *asociatividad por conjuntos* tiene más sentido para las cachés de segundo nivel. Para sacar conclusiones, a continuación se muestra un ejemplo.

Ejemplo que muestra el impacto de la asociatividad de la caché de segundo nivel en la penalización de fallos. Dados los datos siguientes:

- La asociatividad por conjuntos de dos vías, incrementa el tiempo de aciertos en un 10 % de un ciclo de reloj de la CPU
- Tiempo de aciertos$_{L2}$ para correspondencia directa = 4 ciclos de reloj
- Frecuencia local de fallos$_{L2}$ para correspondencia directa = 25 %
- Frecuencia local de fallos$_{L2}$ para asociativa por conjuntos de dos vías = 20 %
- Penalización de fallos$_{L2}$ = 30 ciclos de reloj

¿Cuál es el impacto de la asociatividad de la caché de segundo nivel en la penalización de fallos?

Solución: Para un sistema con caché de segundo nivel de correspondencia directa, la penalización de fallos de la caché de primer nivel es:

$$Penalización\ de\ \ fallos_{L1} = 4\ ciclos\ de\ reloj + 25\% \cdot 30\ ciclos\ de\ reloj = 11.5\ ciclos\ de\ reloj$$

Si a esto se le añade el coste de la asociatividad, se incrementa el coste de acierto sólo en 0.1 ciclos de reloj (como indica el enunciado), y la nueva penalización de fallos de la caché del primer nivel será:

$$Penalización\ de\ \ fallos_{L1} = 4.1\ ciclos\ de\ reloj + 20\% \cdot 30\ ciclos\ de\ reloj = 10.1\ ciclos\ de\ reloj$$

Hay que tener en cuenta también que, las cachés de segundo nivel están casi siempre sincronizadas con las del primer nivel y con la CPU. Por lo tanto, el tiempo de aciertos del segundo nivel debe ser un número entero de ciclos de reloj. Teniendo suerte, se puede aproximar el tiempo de aciertos del segundo nivel a 4 ciclos; si no, se puede redondear hasta 5 ciclos. En cualquiera de los casos, con 4 ó 5 ciclos, la asociatividad implica una mejora sobre la caché de segundo nivel con correspondencia directa:

$$Penalización\ de\ \ fallos_{L1} = 4\ ciclos\ de\ reloj + 20\% \cdot 30\ ciclos\ de\ reloj = 10.0\ ciclos\ de\ reloj$$

$$Penalización\ de\ \ fallos_{L1} = 5\ ciclos\ de\ reloj + 20\% \cdot 30\ ciclos\ de\ reloj = 11.0\ ciclos\ de\ reloj$$

Siguiendo con el análisis de la memoria caché de segundo nivel, aunque la mayor asociatividad es digna de tener en cuenta por tener un pequeño impacto en el tiempo de aciertos del segundo nivel de caché y por deberse la mayor parte del tiempo medio de acceso a los fallos, para cachés muy grandes, los beneficios de la asociatividad disminuyen por haberse eliminado muchos fallos de conflicto.

En tercer lugar, mientras que la localidad espacial mantiene que puede haber un beneficio al incrementar el *tamaño de bloque*, este incremento puede aumentar los fallos de conflicto en cachés pequeñas (puede no haber suficiente sitio para poner datos), y como consecuencia la frecuencia de fallos. Como esto no es un problema en las cachés de segundo nivel grandes y, por ser nada más que relativamente mayor el tiempo de acceso a memoria, son usuales los tamaños de bloque grandes para la memoria caché de segundo nivel. La Figura 2.22 muestra la variación del tiempo de ejecución en función del tamaño de bloque de la caché de segundo nivel.

Finalmente, en cuarto lugar, nos planteamos si todos los datos que estén en la caché de primer nivel deben estar siempre en la caché de segundo nivel. Si es así, se dice que la caché de segundo nivel tiene la *propiedad de inclusión multinivel*. Esta propiedad es deseable puesto que la consistencia entre E/S y cachés (o entre cachés en un multiprocesador) se puede determinar comprobando la caché del segundo nivel. El inconveniente de esta inclusión natural es que, el tiempo medio de acceso a memoria más bajo de la caché de primer nivel, puede sugerir bloques más pequeños para esta caché (que es más pequeña) y bloques mayores para la caché de segundo nivel (que es mayor). Esta inclusión se puede mantener todavía con poco trabajo extra en un fallo del segundo nivel: la caché del segundo nivel debe invalidar todos los bloques del primer nivel que se correspondan con el bloque del segundo nivel que se va a reemplazar (lo que provoca una frecuencia de fallos del primer nivel ligeramente mayor).

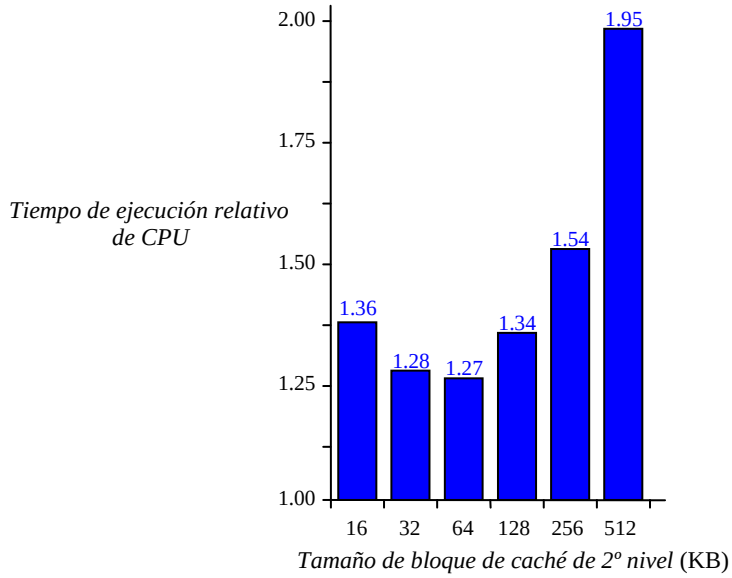


Figura 2.22 Variación del tiempo de ejecución relativo en función del tamaño de bloque de la caché de segundo nivel. Przybylski [1990] recopiló los datos utilizando una caché de segundo nivel de 512 KB de capacidad. Empleó trazas de programas de sistema y usuario del VAX y del MIPS R2000, del libro "*Arquitectura de Computadores. Un enfoque cuantitativo*", de Hennessy y Patterson.

CACHÉ Y E/S. **Reducción de la frecuencia de fallos reduciendo limpiezas de caché**.

Aunque ya no hay mucho más que pueda mejorar el tiempo de ejecución de la CPU, hay algunas cuestiones en el diseño de la memoria caché que mejoran el rendimiento del sistema, particularmente en lo relativo a la E/S.

Como consecuencia de la existencia de cachés, los datos pueden encontrarse tanto en memoria principal como en la memoria caché. Aunque sea la CPU el único dispositivo que lee y modifica los datos, estando la caché entre la CPU y la memoria principal, hay poco peligro de que la CPU vea la copia antigua u obsoleta (*stale*) en principio. Pero, como consecuencia del sistema de E/S, si éste puede acceder directamente a memoria principal, podemos encontrarnos con que los datos que lee la CPU de la memoria caché sean inconsistentes con los de la memoria principal si la E/S los modifica; de la misma forma, si el sistema de E/S lee datos de la memoria principal que han sido modificados únicamente en la memoria caché por la CPU, serán inconsistentes. A este problema se le conoce como *coherencia de caché*.

Hay una pregunta que se debe plantear: ¿a dónde se dirigen los dispositivos de E/S cuando se realizan las operaciones de E/S del computador, a la memoria principal o a la memoria caché?

1. Si los dispositivos de E/S acceden directamente a la memoria caché, tanto estos como la CPU ven los mismos datos; estando resuelto el problema, en principio. Pero, este enfoque presenta problemas:
    a. Los dispositivos de E/S interfieren con la CPU (las E/S provocarán detenciones de la CPU).
    b. La E influye en el rendimiento de la memoria caché al sustituir alguna información por los nuevos datos, que tienen poca probabilidad de ser accedidos por la CPU en un futuro inmediato.
2. Si los dispositivos de E/S se conectan con la memoria principal, se tiene un problema de coherencia.

El objetivo para el sistema de E/S de un computador que tiene el nivel de memoria caché es: evitar el problema de los datos obsoletos, a la vez de interferir lo menos posible con la CPU.

Muchos diseñadores prefieren que las E/S se realicen directamente con la memoria principal (que actúa como buffer de E/S). Utilizando una *caché de escritura directa*, la memoria principal siempre tiene una copia actualizada de la información, eliminando el problema de datos obsoletos en la S (por esta razón, muchas máquinas utilizan escritura directa); aunque las E requerirán algún trabajo extra, este problema puede resolverse:

1. Por software, si éste garantiza que ningún bloque del buffer de E/S designado para E esté en la caché. Para este software, existen dos posibles soluciones:
    a. una primera solución consiste en que se marquen algunas páginas del buffer como “no cacheables”, realizando siempre el sistema operativo sus operaciones de E sobre dichas páginas;
    b. una segunda solución consiste en que el sistema operativo elimine de la caché las direcciones correspondientes al buffer una vez se ha realizado la E.
2. Por hardware, comprobando las direcciones de E/S en la E para ver si están en la caché. Si es así, las entradas correspondientes de la caché se invalidan para evitar datos obsoletos.

Las soluciones planteadas también se pueden utilizar para el caso de S con cachés de postescritura.

COHERENCIA DE CACHÉ EN SISTEMAS MULTIPROCESADORES. **Reducción del tráfico del bus.**

El problema de coherencia de caché se presenta tanto con la E/S como con los multiprocesadores. De forma diferente a la E/S, donde copias múltiples de datos es una situación no común, y a evitar siempre que sea posible, un programa que se ejecuta en un sistema multiprocesador es normal que tenga copias del mismo dato en varias de las cachés asociadas a cada uno de los procesadores de dicho sistema multiprocesador. El rendimiento de un programa en un sistema multiprocesador depende de la eficiencia de dicho sistema para compartir los datos. Los protocolos para mantener la coherencia de múltiples procesadores se denominan *protocolos de coherencia de caché*. Hay dos clases de protocolos para mantener la coherencia de caché:

- *Basados en directorio*. La información sobre un bloque de memoria física se mantiene en una única posición. Hay un único directorio en la memoria principal que contiene el estado de cada bloque. La información de ese directorio puede contener qué cachés tienen copias del bloque, si el bloque está modificado (*dirty*) o no, etc. Las entradas al directorio pueden estar distribuidas para que diferentes peticiones puedan ir a diferentes memorias, reduciendo así la contención. Sin embargo, conservan la característica de que el estado de un bloque compartido está siempre en una única posición conocida.
- *Espionaje (snooping)*. Cada caché que tiene una copia de datos de un bloque de memoria física también tiene una copia de la información sobre él. Estas cachés se utilizan habitualmente en sistemas de memoria compartida con un bus común; todos los controladores de caché vigilan o espían (snoop) en el bus para determinar si tienen o no una copia del bloque compartido. Estos protocolos se hicieron populares con los sistemas multiprocesadores que utilizaban microprocesadores y cachés, en sistemas de memoria compartida, por poder utilizar una conexión física preexistente: el bus de memoria. Esta clase de protocolo tiene una ventaja sobre los basados en directorio: la información de coherencia es proporcional al número de bloques de la memoria caché, en lugar de ser proporcional al número de bloques de la memoria principal. Como contrapartida, los directorios no requieren un bus que vaya a todas las cachés y, por consiguiente, puede ser más escalable un sistema (más procesadores).

El problema de coherencia se reduce a que un procesador tenga acceso exclusivo al escribir un objeto y a que tenga la copia más reciente al leer un objeto. Por tanto, ambos protocolos deben localizar todas las cachés que comparten el objeto que se va a escribir. La consecuencia de una escritura en un dato compartido es *invalidar* las demás copias o *difundir* la escritura a las copias compartidas. Por existir cachés de postescritura, los protocolos de coherencia de caché también deben ayudar a determinar, en los fallos de lectura, quién tiene el valor más actualizado.

Para los protocolos de espionaje, a los bits de estado que ya existen en un bloque, se les añade información sobre la compartición; información que se utiliza para vigilar las actividades del bus. En un fallo de lectura, todas las cachés comprueban si tienen una copia del bloque requerido y realizan la acción apropiada (por ejemplo puede suministrar el dato a la caché donde se produjo el fallo. Análogamente, en una escritura todas las cachés comprueban si tienen una copia y actúan adecuadamente, invalidando su copia o cambiándola por el nuevo valor.

Como en cada transacción del bus se comprueban las etiquetas de dirección de la caché, se podría suponer que interfiere con la CPU. Esto sería efectivamente así si no se duplicase la parte de etiqueta de dirección de la caché (no la caché completa) para, de esa forma, obtener un puerto extra de lectura para espiar (*snooping*). De esta forma, solamente interfiere el espionaje con el acceso de la CPU a la caché cuando hay un problema de coherencia (también, con espionaje, en un fallo la CPU debe arbitrar con el bus el cambio de las etiquetas de espionaje además de las normales). Cuando hay un problema de coherencia de caché, se detendrá la CPU por estar indisponible la caché. En cachés multinivel, si la comprobación de coherencias se puede limitar a la de nivel inferior a causa de inclusión multinivel, probablemente no será necesario duplicar las etiquetas de dirección.

Dependiendo de lo que ocurra en una escritura, los *protocolos de espionaje* pueden ser de dos tipos:

- *De invalidación en escritura*. El procesador que escribe hace que se invaliden todas las copias en el resto de cachés antes de cambiar la copia de su caché; teniendo libertad para actualizar el dato hasta el instante en que otro procesador lo pida. El procesador que escribe, distribuye una señal de invalidación sobre el bus, y todas las cachés comprueban si tienen una copia; si es así, deben invalidar el bloque que contenga la palabra → Este esquema permite múltiples lectores pero sólo un escritor.
- *De difusión en escritura*. En lugar de invalidar cada copia del bloque compartido, el procesador que escribe, difunde el nuevo dato sobre el bus; entonces se actualizan todas las copias con el nuevo valor. Los protocolos de difusión en escritura permiten habitualmente que los bloques se identifiquen como *compartidos* (difundidos) o *privados* (locales). Este protocolo actúa de forma similar a una caché de escritura directa para *datos compartidos* (difundiendo a otras cachés) y como una caché de postescritura para *datos privados* (los datos modificados salen fuera de la caché sólo cuando se produce un fallo).

La mayoría de los sistemas multiprocesadores con memoria caché, utilizan cachés de postescritura porque reducen el tráfico del bus y, de esa forma, se permiten más procesadores sobre un solo bus. Las cachés de postescritura utilizan *invalidación* o *difusión*, existiendo numerosas variaciones para ambas alternativas. Hasta ahora no hay consenso sobre cuál es el esquema mejor; algunos programas tienen menos gastos de coherencia con invalidación de escritura y otros con difusión de escritura.

Como se ha visto, el tamaño de bloque juega un papel importante en la coherencia de caché. Tomando como ejemplo el caso de espiar una caché de segundo nivel con un tamaño de bloque de ocho palabras, y que una palabra sea escrita y leída alternativamente por dos procesadores; tanto si se utiliza invalidación como difusión en escritura, el protocolo que sólo difunda o envíe una palabra tiene ventaja sobre un esquema que transfiera el bloque completo. Otro aspecto relacionado con los grandes bloques es la *compartición falsa*: dos variables diferentes compartidas están localizadas en el mismo bloque de caché, haciendo que se transfiera el bloque entre procesadores aún cuando estos estén accediendo a variables diferentes (normalmente pasa esto por un mal diseño del tamaño de las páginas y por la poca relación existente entre variables de la misma página.

Medidas actuales indican que los *datos compartidos* tienen menor localidad espacial y temporal que la observada para otro tipo de datos, independientemente de la política de coherencia.

Ejemplo de protocolo

Para ilustrar la complejidad de un protocolo de coherencia de caché, la Figura 2.23 muestra un *diagrama de transición de estados finitos para un protocolo de invalidación en escritura basado en una política de postescritura*. Los tres estados del protocolo están duplicados para que representen transiciones basadas en acciones de la CPU, en contraposición a las transiciones basadas en las operaciones del bus. Esto se hace sólo para esta figura; sólo hay una máquina de estados finitos, por memoria caché, con estímulos provenientes o de la CPU conectada que corresponda o del bus.

Las transiciones se presentan en los fallos de lectura, fallos de escritura o aciertos de escritura; los aciertos de lectura no hacen cambiar el estado de la caché. Cuando la CPU tiene un fallo de lectura, cambia el estado de ese bloque a *sólo lectura* y postescribe el bloque antiguo si estaba en el estado *lectura/escritura* (modificado). Todas las cachés comprueban, a raíz del fallo de la lectura, si este bloque está en su caché. Si se tiene una copia y está en el estado de *lectura/escritura*, entonces el bloque se escribe en memoria y se cambia al estado de *inválido*. (Una optimización no mostrada en la figura, realizaría el cambio de estado de ese bloque a sólo lectura). Cuando una CPU escribe en un bloque, ese bloque va al estado de *lectura/escritura*. Si la escritura fuese un acierto, una señal de invalidación recorrería el bus. Como las cachés vigilan el bus, todas comprueban si tienen una copia de ese bloque; si la tienen, lo invalidan. Si la escritura fuese un fallo, todas las cachés con copias irían al estado inválido.

Como se puede imaginar, hay muchas variaciones en la coherencia de caché que son mucho más complicadas que este modelo sencillo. Las variaciones incluyen que otras cachés intenten o no suministrar el bloque si tienen una copia, si el bloque debe o no debe ser invalidado en un fallo de lectura, así como invalidar la escritura en vez de difundirla como se ha explicado antes. La Tabla 2.7 resume algunos protocolos de espionaje de coherencia de caché.

Nombre	**Categoría**	**Política de escritura de memoria**	**Característica única**
Write Once	Invalidación en escritura	Postescritura después de primera escritura	
Synapse N+1	Invalidación en escritura	Postescritura	Posesión de memoria explícita
Berkeley	Invalidación en escritura	Postescritura	Estado compartido propio
Illinois	Invalidación en escritura	Postescritura	Estado privado limpio; puede suministrar datos de cualquier caché con una copia limpia
Firefly	Difusión en escritura	Postescritura para privada, Escritura directa para compartida	Memoria actualizada en difusión
Dragon	Difusión en escritura	Postescritura para privada, Escritura directa para compartida	Memoria noactualizada en difusión

Tabla 2.7 Resumen de seis protocolos de espionaje. Archibald y Baer [1986] utilizan estos nombres para describir los seis protocolos, y Eggers [1989], resume las analogías y diferencias de la forma mostrada en la tabla.

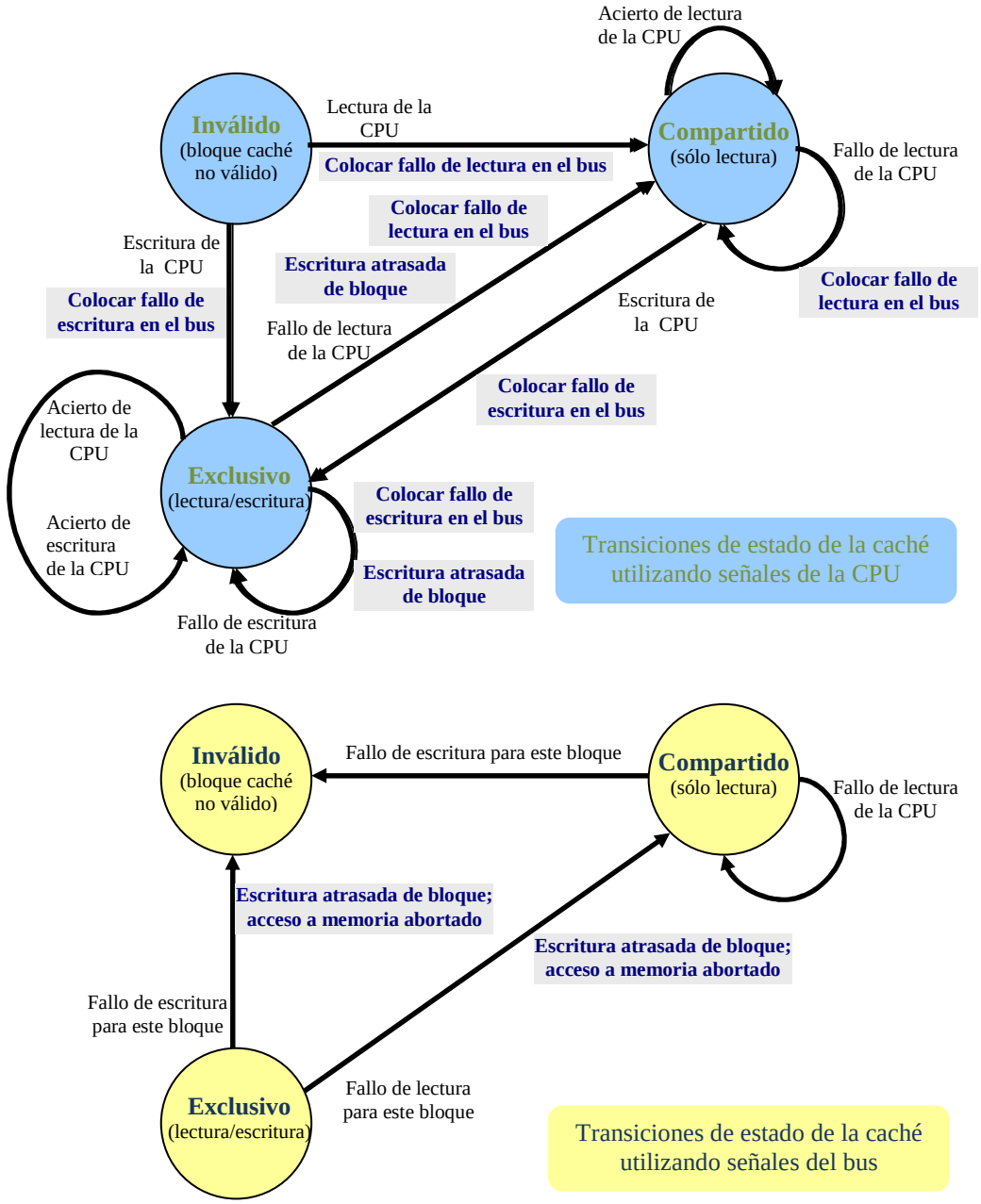


Figura 2.23 Ejemplo de protocolo de coherencia de caché de invalidación en escritura para una caché de postescritura. La parte superior del diagrama muestra las transiciones entre estados basadas en acciones de la CPU asociada con esta caché; la parte inferior muestra las transiciones basadas en operaciones sobre el bus. Sólo hay una máquina de estados en una caché, aunque se representen dos para aclarar cuándo ocurre una transición. Los estados de la caché se muestran con círculos, mostrando su nombre entre paréntesis; los estímulos causan un cambio de estado, y se muestra en los arcos de transición en letra normal; cualquier acción del bus generada como parte de la transición de estado se muestra en el arco de transición en un cradro gris y letra negrita.

Sincronización utilizando coherencia

Uno de los principales requerimientos de un sistema multiprocesador de memoria compartida es que pueda coordinar procesos que trabajen sobre una tarea común. Normalmente, un programador utilizará *variables de bloqueo* para sincronizar los procesos.

La dificultad para un arquitecto de un sistema multiprocesador está en proporcionar un mecanismo para decidir qué procesador supera el bloqueo y proporcionar la operación que bloquea sobre una variable. La arbitración es fácil para los sistemas multiprocesadores de bus compartido, ya que el bus es el único camino a memoria: el procesador que consigue el bus, bloquea el acceso a memoria al resto de procesadores. Si la CPU y el bus proporcionan una *operación atómica de intercambio*, los programadores pueden crear bloqueos con la semántica apropiada. El adjetivo atómica es clave; significa que un procesador puede leer una posición e inicializar el valor bloqueado en la misma operación del bus, evitando que cualquier otro procesador lea o escriba en memoria.

La Figura 2.24 muestra un procedimiento típico para bloquear una variable utilizando una instrucción de intercambio atómica. Un procesador lee primero la variable de bloqueo para examinar su estado. Un procesador sigue leyendo y examinando hasta que el valor indique que el bloqueo ha desaparecido. El procesador compite entonces con los demás procesadores que, análogamente, están en el "bucle de espera" para ver quién puede bloquear primero la variable. Todos los procesos utilizan una instrucción de intercambio que lee el valor antiguo y almacena un **1** en la variable de bloqueo. El ganador verá el **0**, y los perdedores verán el **1** que colocó allí el ganador. (Aunque los perdedores a continuación inicializarán la variable con el valor de bloqueado, no importa). El procesador ganador ejecuta el código después del bloqueo y, después, cuando termina, almacena un **0** en la variable de bloqueo, comenzando todos nuevamente la carrera. La instrucción de intercambio atómica "*examina e inicializa*" (*test and set*) se denomina así en algunos repertorios de instrucciones porque primero examina el valor antiguo y después lo inicializa.

El esquema de "bloqueo circular" (*spin lock*) con coherencia de caché basado en bus, mostrado en la Tabla 2.8, se examina a continuación.

Paso	Procesador P0	Procesador P1	Procesador P2	Actividad bus
1	Tiene bloqueo	Gira, examinando si bloqueo = **0**	Gira, examinando si bloqueo = **0**	Ninguna
2	Pone bloqueo a **0** y envía **0** al bus			Invalida variable de bloqueo de P0
3		Fallo de caché	Fallo de caché	Bus decide servir fallo de caché de P2
4		Espera mientras bloqueo ocupado	Bloqueo = **0**	Fallo de caché para P2 satisfecho
5		Bloqueo = **0**	Intercambio: lee bloqueo y lo pone a **1**	Fallo de caché para P1 satisfecho
6		Intercambio: lee bloqueo y lo pone a **1**	Valor de intercambio = **0** y envía **1** al bus	Invalida variable de bloqueo de P2
7		Valor de intercambio = **0** y envía **1** al bus	Entra sección crítica	Invalida variable de bloqueo de P1
8		Gira examinando si bloqueo = **0**		Ninguna

Tabla 2.8 Pasos dados en el esquema de bloqueo circular y coherencia de caché basada en bus, para tres procesadores P0, P1 y P2. Se supone coherencia de invalidación de escritura. P0 arranca con el bloqueo (paso 1). P0 sale y desbloquea el bloqueo (paso 2). P1 y P2 corren para ver quién lee el valor desbloqueado durante el intercambio (paso 3-5). P2 gana y entra en la sección crítica (pasos 6 y 7), mientras que P1 da vueltas y espera (pasos 7 y 8).

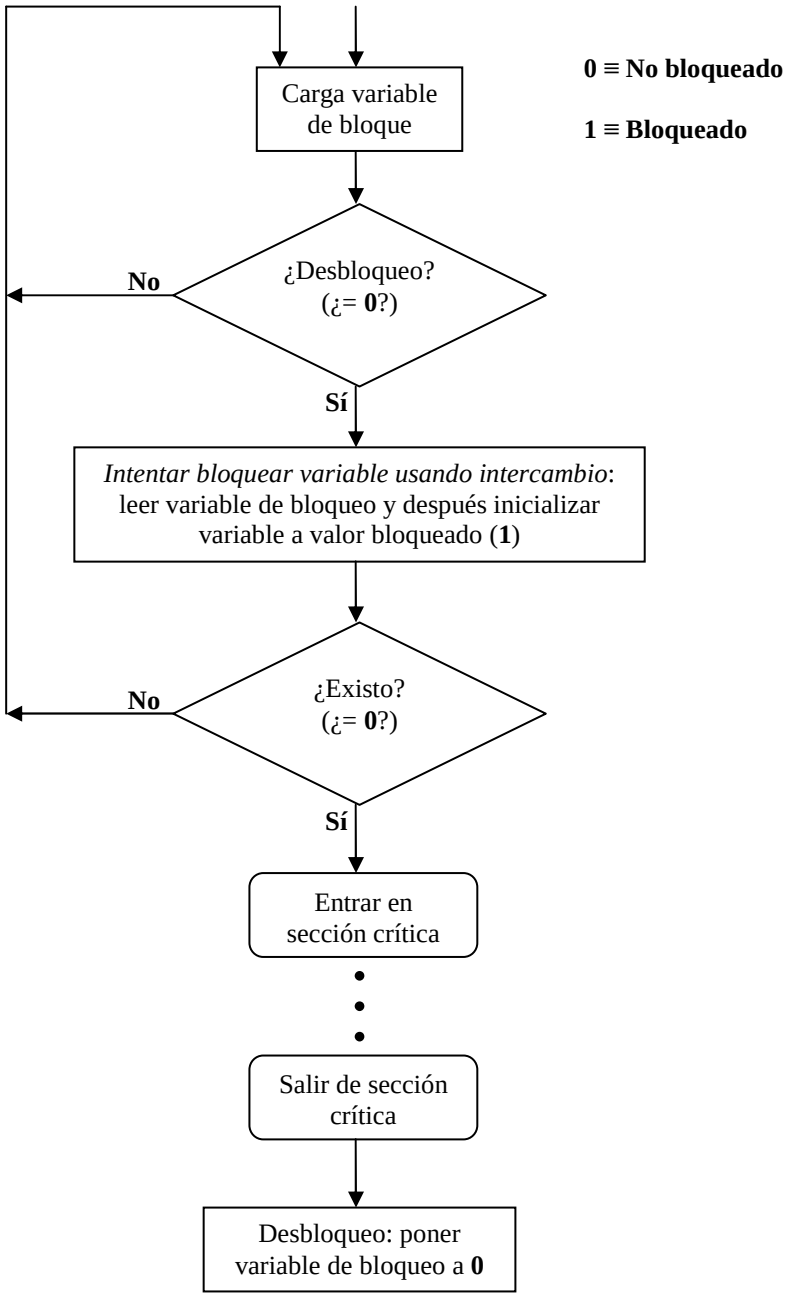


Figura 2.24 Pasos para superar un bloqueo para sincronizar procesos y después dejar el acceso desbloqueado a la salida de la sección clave del código.

Una ventaja de este algoritmo es que permite que los procesadores estén en una situación de bucle de espera sobre una copia local de la variable de bloqueo en sus cachés. Esto reduce la cantidad de tráfico del bus en comparación con los algoritmos de bloqueo que emplea un bucle que intenta realizar un examen e inicialización (*test and set*). (La tabla 2.8 muestra las operaciones del bus y de la caché para múltiples procesadores intentando bloquear una variable). Una vez que el procesador que superó el bloqueo almacena un **0** en la variable de bloqueo, todas las demás cachés ven ese almacenamiento e invalidan su copia de la variable de bloqueo. Entonces obtienen el nuevo valor, que es **0**. (Con coherencia de caché de difusión en escritura, las cachés actualizarán su copia en lugar de invalidarla primero y cargarla después de memoria). Este nuevo valor da la salida a la carrera para ver quién puede inicializar

primero la variable de bloqueo. El ganador consigue el bus y almacena un **1** en la variable de bloqueo; las otras cachés sustituyen su copia de la variable de bloqueo, que contiene **0**, por un **1**. A partir de aquí, leen que la variable está ya bloqueada y deben volver a examinar e iterar. Este esquema tiene dificultades para escalar a muchos procesadores a causa del tráfico de cominicaciones generado cuando desaparece el bloqueo.

Modelos de consistencia de memoria

Cuando se introduce coherencia de caché para mantener la consistencia de múltiples copias de un objeto, surge una nueva pregunta: ¿qué consistencia deben tener los valores vistos por dos procesadores?. Para comprender mejor el problema, se muestra un ejemplo: supongamos los segmentos de código de dos procesos P1 y P2:

```
P1:   A = 0;                          P2:   B = 0;
      . . . . .                             . . . . .
      A = 1;                                B = 1;
L1:   if (B = = 0) . . .        L2:   if (A = = 0) . . .
```

Suponer que los procesos están siendo ejecutados en procesadores diferentes y que las posiciones A y B están originalmente “cacheadas” por ambos procesadores con el valor inicial de **0**. Si la memoria siempre es consistente, será imposible para ambas sentencias “if” (etiquetas L1 y L2) evaluar sus condiciones como verdaderas (bien A = **1** o B = **1**). Pero, si se supone que las invalidaciones de escritura tienen un retardo, y que al procesador se le permite continuar durante este retardo, hace que sea posible que P1 y P2 no hayan visto las invalidaciones para B y A (respectivamente) antes de que intenten leer los valores. La pregunta que surge con este ejemplo es: ¿cuán consistente debe ser la visión que tengan de memoria diferentes procesadores?

Existen varias propuestas:

- La llamada *consistencia secuencial*, requiere que el resultado de cualquier ejecución sea el mismo que si los accesos de cada procesador se mantuviesen en orden y los accesos entre diferentes procesadores se intercalasen arbitrariamente. En este caso, la aparente anomalía del ejemplo anterior no se puede presentar. Implementar consistencia secuencial requiere, habitualmente, que un procesador retarde cualquier acceso a memoria hasta que se completen todas las invalidaciones provocadas por las escrituras anteriores. Aunque este modelo presenta un sencillo paradigma de programación, reduce el rendimiento potencial, especialmente en máquinas con un gran número de procesadores o en máquinas con grandes retardos de interconexión.
- Otros modelos con *consistencia de memoria más débil* (por ejemplo, requerir al programador para que utilice instrucciones de sincronización para ordenar accesos a memoria a la misma variable), en lugar de retardar todos los accesos hasta que se completen las invalidaciones, sólo necesitan que se retrasen los accesos de sincronización.

El ejemplo propuesto funcionará con consistencia secuencial, pero no con un modelo más débil. Para que una consistencia débil produzca los mismos resultados que la consistencia secuencial, el programa tendría que modificarse para que incluyese operaciones de sincronización que ordenasen los accesos a las variables A y B. Algunas máquinas eligen implementar la *consistencia secuencial* como modelo de programación, mientras que otras optan por una *consistencia más débil*. Cuantos más procesadores tiene el sistema, el problema de la consistencia es más crítico para el rendimiento.

2.6 OBSERVACIONES FINALES

La dificultad de construir un sistema de memoria con las mismas prestaciones que las CPU más rápidas está acentuada por el hecho de que la materia prima de la memoria principal es la misma que se encuentra en los computadores más baratos. El *principio de localidad* es el que en cierto modo salva del problema a dichos sistemas.

Los fallos de cada nivel de jerarquía se pueden categorizar según tres causas: *forzosos*, por *capacidad* y por *conflicto*; empleándose para cada caso, para paliarlos, diferentes técnicas.

Suele haber un codo en la *curva de coste/rendimiento* de la jerarquía de memoria: antes del codo se desperdicia rendimiento y después del codo se malgasta hardware. Los arquitectos de computadores encuentran ese punto por simulación y análisis cuantitativo.