Arquitectura de Computadores

7. Memoria Caché

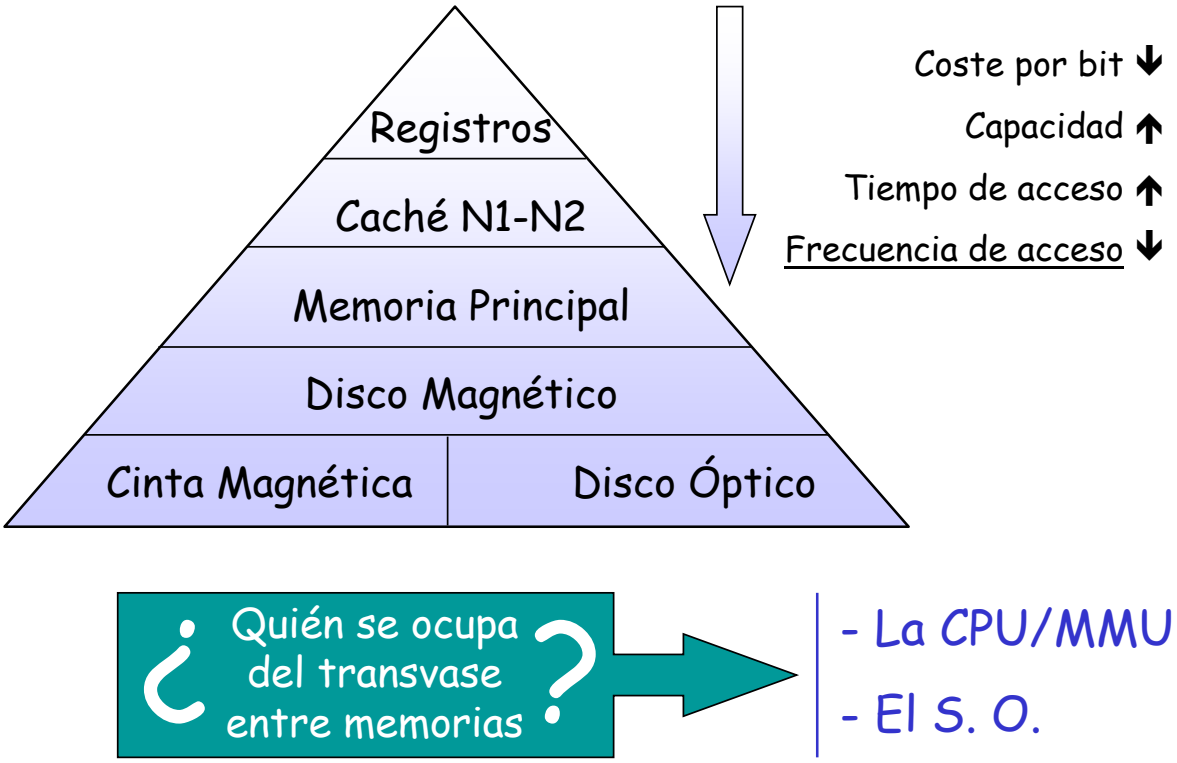
1. Fundamento e Introducción
2. Estructura
3. Políticas de Ubicación
4. Políticas de Sustitución
5. Políticas de Escritura
6. Otras Consideraciones

En el capítulo anterior comenzamos a ver cómo mejorar las prestaciones de un ordenador convencional. Lo hicimos empezando por ver la forma de aumentar el ritmo de ejecución de instrucciones en la CPU.

Ahora vamos a ocuparnos de cómo mejorar el tiempo de acceso a los datos que están en memoria principal. Para ello incorporaremos la **memoria caché**, un nuevo nivel en la jerarquía de memorias situado entre la CPU y la memoria principal.

Veremos cómo una memoria tan pequeña (en comparación con la memoria principal) puede ser tan útil, y las maneras en que se puede organizar para aprovecharla de la mejor manera posible.

Jerarquías de Memoria



Como vimos en el capítulo de la memoria principal, un ordenador debe construirse con diversos niveles de memoria organizados en una jerarquía, de tal manera que se pueda obtener, como conjunto, una memoria de mucha capacidad, rápida y barata.

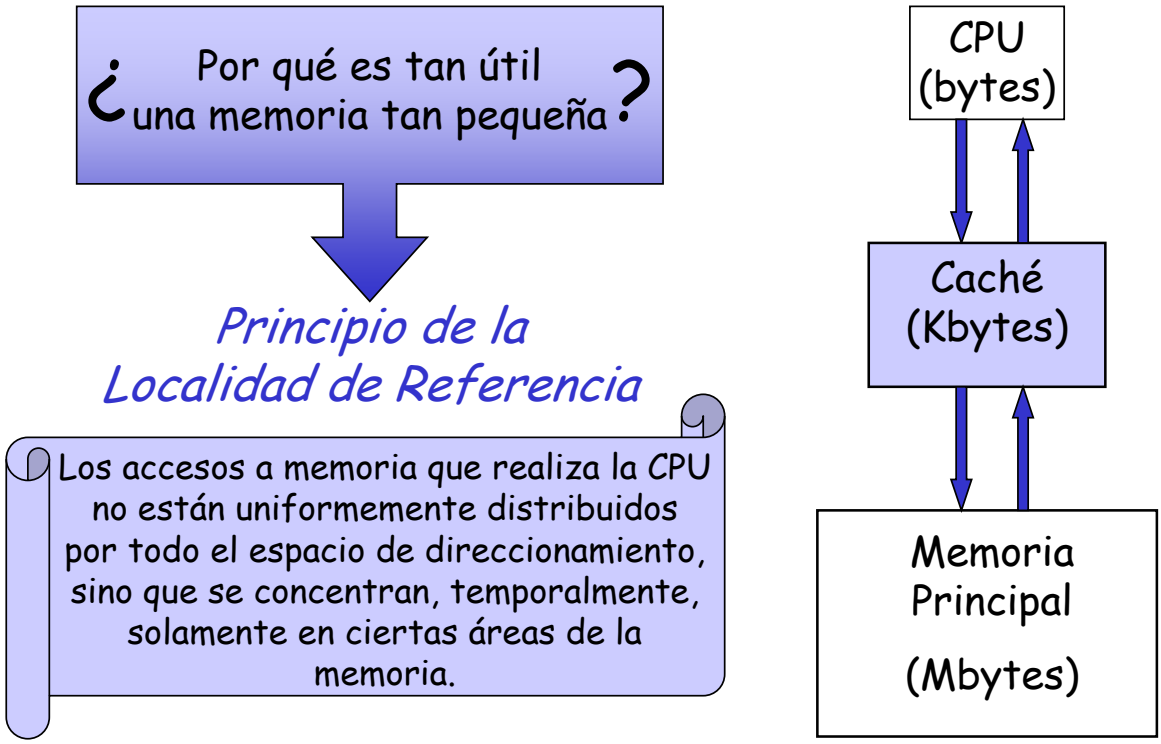
Recordemos la pirámide de memoria en la que, a medida que se va de arriba hacia abajo, sucede lo siguiente:

- Disminuye el coste por bit
- Aumenta la capacidad
- Aumenta el tiempo de acceso
- Disminuye la frecuencia de acceso a la memoria desde la CPU

La clave de la solución está en este último punto: la decreciente frecuencia de acceso. Esto simplemente quiere decir que no se accede a todos los datos con la misma frecuencia; obviamente se accede más a los datos del programa en ejecución que a los de uno que no se ejecuta desde hace un año; y de igual manera, en un momento dado se accede más a los datos de una expresión que se está evaluando en ese preciso instante que a otros datos del programa.

En este capítulo vamos a ocuparnos de la memoria caché. Comenzaremos por ver cómo una memoria tan pequeña (en comparación con la memoria principal) puede ser tan útil.

Fundamento



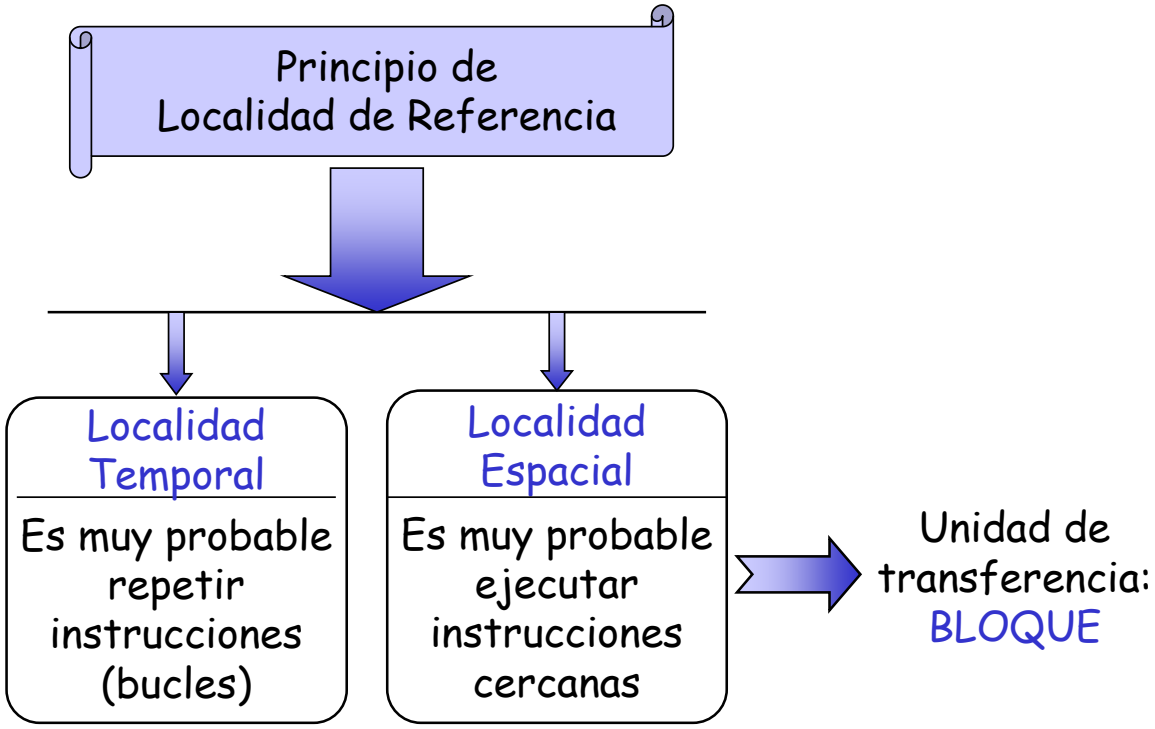
La efectividad del mecanismo de la memoria caché se basa en una propiedad de los programas denominada “localidad de referencia” y que comentamos a continuación.

El **Principio de Localidad de Referencia** dice que los accesos a memoria que realiza la CPU no están uniformemente distribuidos por todo el espacio de direcciones, sino que, temporalmente, se concentran en áreas de la memoria.

Lo que establece este principio se debe a que el contenido de cada programa no está esparcido por toda la memoria, sino que sus instrucciones y datos están contenidos en una o varias secciones de memoria contigua, por lo que los accesos a la memoria se concentran en las áreas donde están las instrucciones o los datos del programa en ejecución.

El análisis de los programas muestra que la mayoría del tiempo de ejecución se dedica a rutinas en las que una serie de instrucciones se ejecutan repetidamente. Estas instrucciones pueden estar formando parte de un bucle, bucles anidados, o unos cuantos procedimientos a los que se llama iterativamente. Por esto, las referencias a memoria en una porción de tiempo dada, se concentran, concretamente, no en todo el programa que se está ejecutando, sino, más específicamente, en el fragmento del bucle que en ese momento se está ejecutando.

Según esto, si el segmento activo de un programa se puede ubicar en una memoria rápida, aunque pequeña, el tiempo total de ejecución puede verse drásticamente reducido.



El principio de localidad se manifiesta en dos aspectos: temporal y espacial.

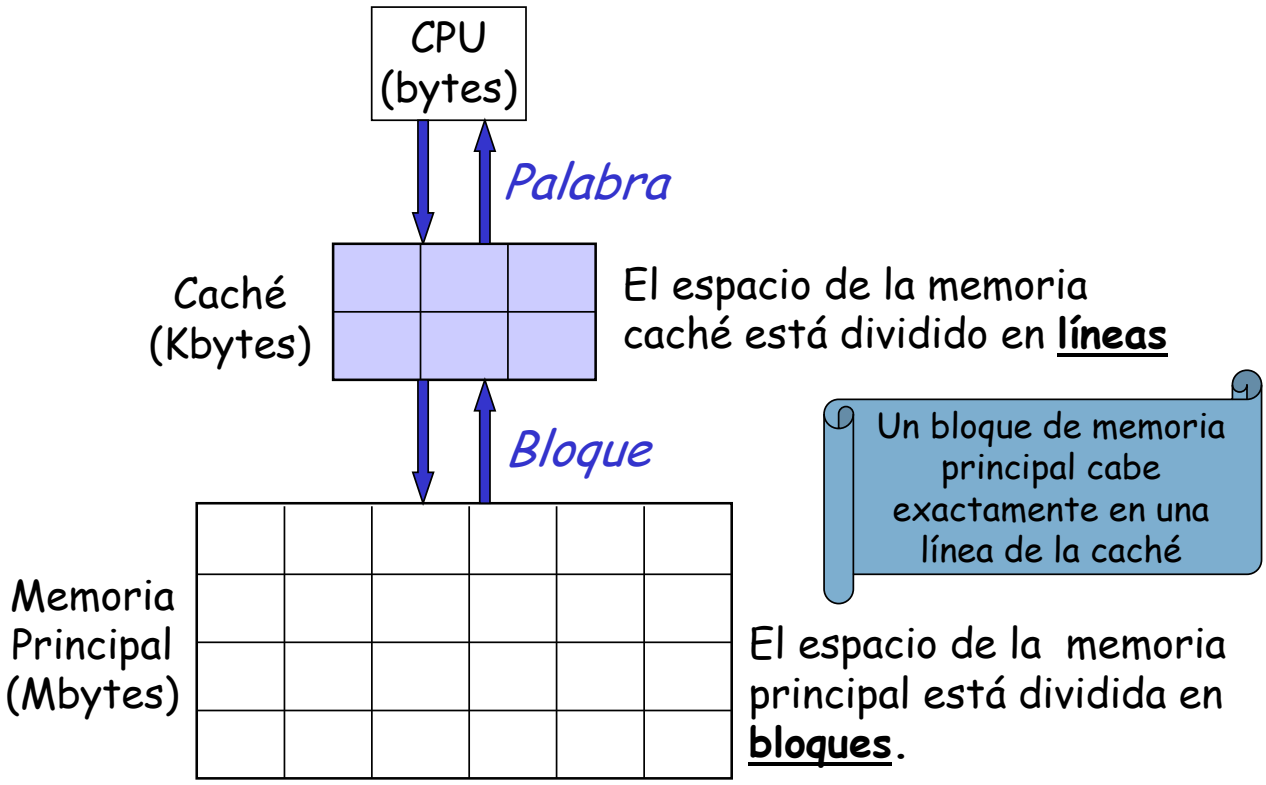
La localidad de referencia **temporal** se debe a la naturaleza repetitiva de los programas o de porciones de los programas, y significa que, tal y como hemos visto en la página anterior, una instrucción que se acaba de ejecutar recientemente es muy probable que se vuelva a ejecutar en un futuro muy próximo. Ahora, la secuencia detallada de la ejecución de las instrucciones no es significativa, lo que realmente nos importa es que muchas instrucciones localizadas en áreas concretas del programa, debido a los bucles que las encierran, se ejecutan repetidamente durante un periodo de tiempo, mientras que al resto del programa o del resto de la memoria solo se accede muy de vez en cuando.

El aspecto **espacial** quiere decir que las instrucciones que se encuentran en la proximidad de una instrucción recientemente ejecutada (en cuanto a sus direcciones en memoria) también es muy probable que se ejecuten muy pronto. A los datos u operandos de un programa les sucede lo mismo, no están dispersos por toda la memoria, sino que están agrupados en una zona de la misma, y ocupando direcciones contiguas.

Dado que el tiempo de acceso a la memoria principal es muy costoso y teniendo en cuenta lo que nos dice la localidad espacial, parece conveniente que cuando se accede a ella para obtener un byte o una palabra necesitada por la CPU, en lugar de transferir solamente el dato solicitado por la CPU, se aproveche “el viaje” para transferir, no solamente ese byte o palabra, sino un bloque contiguo de información que contenga dicho byte y unos pocos más que estén en las direcciones cercanas. Así, cuando se utiliza una memoria caché, la unidad de transferencia entre ésta y la memoria principal es el **bloque**.

El tamaño de un bloque suele ser el ancho del bus de datos o un múltiplo de él, es decir, 2, 4, 8, … bytes.

Introducción

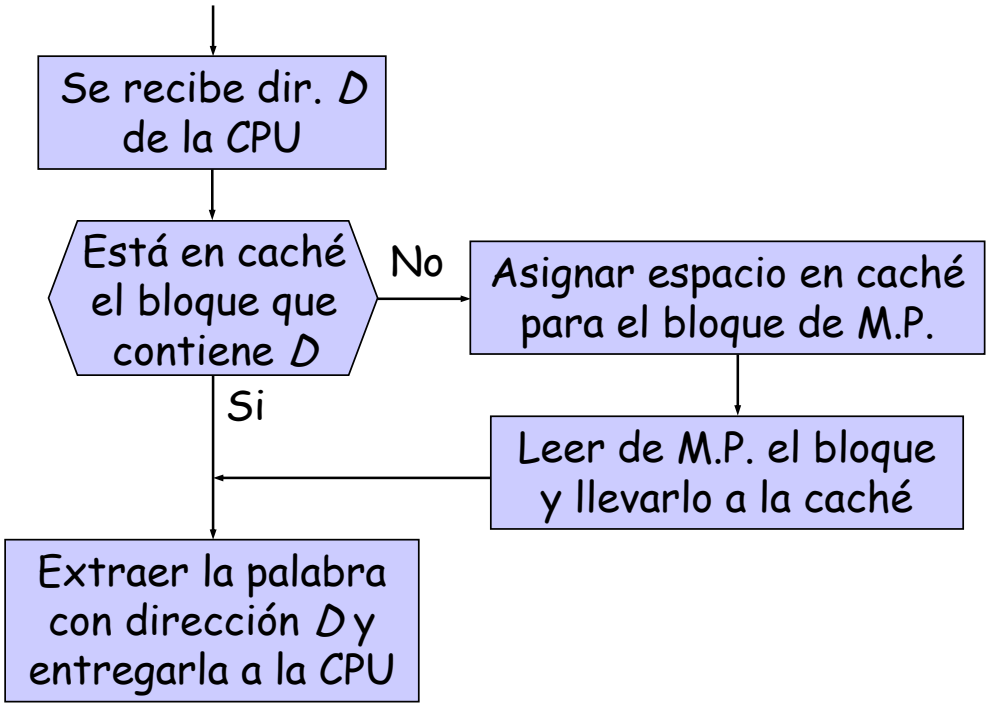


Conceptualmente el funcionamiento de una memoria caché es muy simple. La circuitería de control de la memoria está diseñada para aprovechar la localidad de referencia. De acuerdo con el aspecto temporal de la localidad de referencia, siempre que un dato se necesita por primera vez, se debe traer a la memoria caché, en la que permanece, de tal forma que cuando se le vuelva a necesitar, se podrá obtener muy rápidamente.

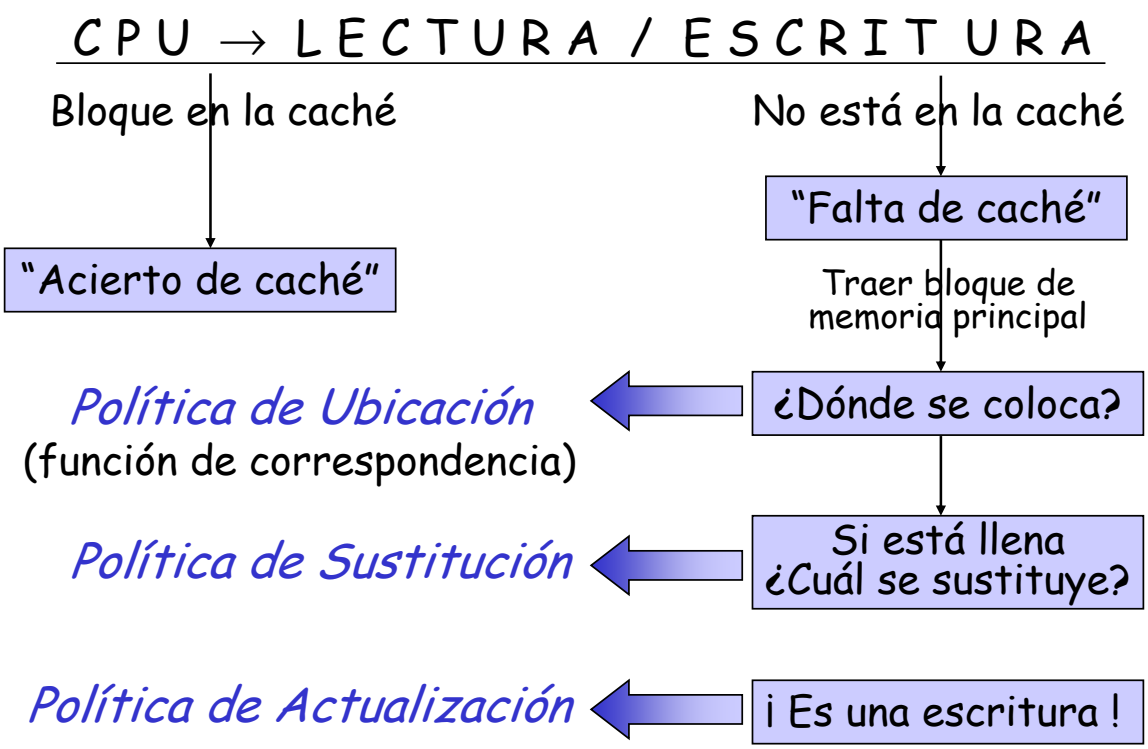
Por su parte, la localidad espacial aconseja que en lugar de traer a la caché solamente el dato referenciado, se traiga el grupo de datos que reside en direcciones adyacentes a dicho dato. En el contexto de las cachés, se emplea el término **bloque** para referirse a un conjunto de datos con direcciones contiguas que se utiliza como unidad de transferencia entre la memoria principal y la caché. El espacio que tiene una caché para albergar un bloque de memoria principal se denomina **línea**.

Como se muestra en la figura, cuando se recibe una petición de lectura de la CPU, el contenido del bloque completo de memoria principal que contiene la dirección especificada se transfiere a la caché. Posteriormente, cuando el programa referencia cualquiera de las direcciones del bloque, el contenido correspondiente se lee directamente de la caché (solamente la palabra o parte de la palabra referenciada, no el bloque completo).

Operación de lectura en un sistema con caché



Cuando la CPU intenta leer una palabra de memoria principal, se le presenta la dirección a la memoria caché y ésta comprueba si tiene el dato correspondiente a esa dirección. Si es así, entrega el dato; si no, se produce una falta de caché, con lo que hay que acudir a la memoria principal para leer un **bloque** de varias palabras adyacentes en memoria. Como ya hemos comentado, el hecho de leer un bloque, y no solamente la palabra concreta referenciada, se debe al principio de localidad de referencia espacial, pues cuando en la CPU se genera una referencia a una palabra concreta, es muy fácil que en las próximas lecturas se haga referencia a otras palabras cercanas a esa palabra.



Como se muestra en la figura, cuando se recibe una petición de lectura de la CPU, el contenido del bloque completo de memoria principal que contiene la dirección especificada se transfiere a la caché. Posteriormente, cuando el programa referencia cualquiera de las direcciones del bloque, el contenido correspondiente se lee directamente de la caché.

Normalmente la memoria caché puede almacenar un número de bloques de memoria que es muy pequeño comparado con el número de bloques de la memoria principal. La correspondencia entre los bloques de la memoria principal y los que se encuentran en la caché se denomina "función de correspondencia". La función de correspondencia a utilizar se establece mediante la **política de ubicación**. Cuando la caché está llena, y se referencia una dirección cuyo bloque no está en la caché, el hardware de la caché debe decidir qué bloque deberá ser expulsado para dejar espacio al bloque que se acaba de referenciar. El conjunto de reglas que se utilizan para tomar esta decisión se denominan algoritmos o **políticas de sustitución**.

Obsérvese que la CPU no necesita conocer explícitamente la existencia de la caché, sino que genera una petición de lectura o escritura utilizando direcciones de memoria principal, y la circuitería de la caché determina si la palabra solicitada está o no está en ese momento en la caché. Si está (se dice que se ha producido un acierto de caché), la operación de lectura o escritura se realiza sobre la dirección correspondiente en la caché, y si la operación es de lectura, la memoria principal no se ve implicada en la ejecución. Si la operación es una escritura, la palabra afectada en la memoria caché deberá actualizarse, en algún momento, en la memoria principal, para que el contenido de un bloque en la caché sea idéntico al de su correspondiente en memoria principal. En estos casos, la actualización en memoria principal de los bloques modificados en la caché se realiza según una **política de escritura o de actualización**.

Cuando el dato referenciado por una lectura de la CPU no está en la caché, se produce una falta o fallo de caché. En este caso, el bloque que contiene la palabra referenciada se copia de la memoria principal a la caché, y a continuación, la palabra solicitada se envía a la CPU. Otra opción consiste en enviarle la palabra directamente de la memoria principal a la CPU y al mismo tiempo cargarla en la caché. Esta última técnica, denominada "carga directa" (*load through),* aunque reduce el tiempo de espera de la CPU, requiere un hardware más complejo.

Más adelante, al comentar las políticas de escritura, veremos las opciones a tomar cuando se produce una falta de caché en una operación de escritura.

En la operación de la caché se nos plantean, por tanto, estas tres preguntas:

P1: ¿Dónde situar un bloque en la caché? (Política de ubicación).

P2: ¿Qué bloque reemplazar ante una caché llena? (Política de sustitución).

P3: ¿Qué hacer ante una operación de escritura? (Política de actualización).

Efectividad de la Memoria Caché

Tiempo medio de acceso a memoria

$$T_{acceso} = T_C \cdot P_A + T_{MP} \cdot (1 - P_a)$$

P_A : Probabilidad de acierto

T_C : Tiempo de acceso a caché

T_{MP} : Tiempo de acceso a M. P.

EJEMPLO

$T_{MP} = 500\ \eta s$

$T_C = 50\ \eta s$

$P_A = 0{,}99$

$$T_{acceso} = 50 \cdot 0{,}99 + 500 \cdot 0{,}01 = 54{,}5\ \eta s$$

¿ Merece la pena la caché ?

$$\text{Índice de mejora} = \frac{T_{sin\ caché}}{T_{con\ caché}} = \frac{500}{54{,}4} = 9{,}19$$

Hemos comentado que cuando se hace referencia a una palabra cuyo bloque está en la caché, tenemos un acierto de caché, mientras que si no lo está, se produce una falta de caché.

Ya sabemos que cuando se produce un acierto de caché, el tiempo de acceso a la palabra referenciada es mucho menor que si hubiera que traerla desde memoria principal, pero si se produce una falta de caché, el tiempo de acceso puede ser mayor que en un sistema de memoria sin caché. (Depende de si tiene carga directa o no). Por esto, se debe evaluar el comportamiento o efectividad del diseño de la caché para saber si el tiempo medio de acceso a memoria realmente mejora y justifica la presencia de la memoria caché.

Veamos un ejemplo. Supongamos una memoria principal con un tiempo de acceso de 500 ηs, y una caché con un tiempo de acceso de 50 ηs, y con una tasa de aciertos del 99%.

¿Cuál es el tiempo medio de acceso a memoria en este sistema?

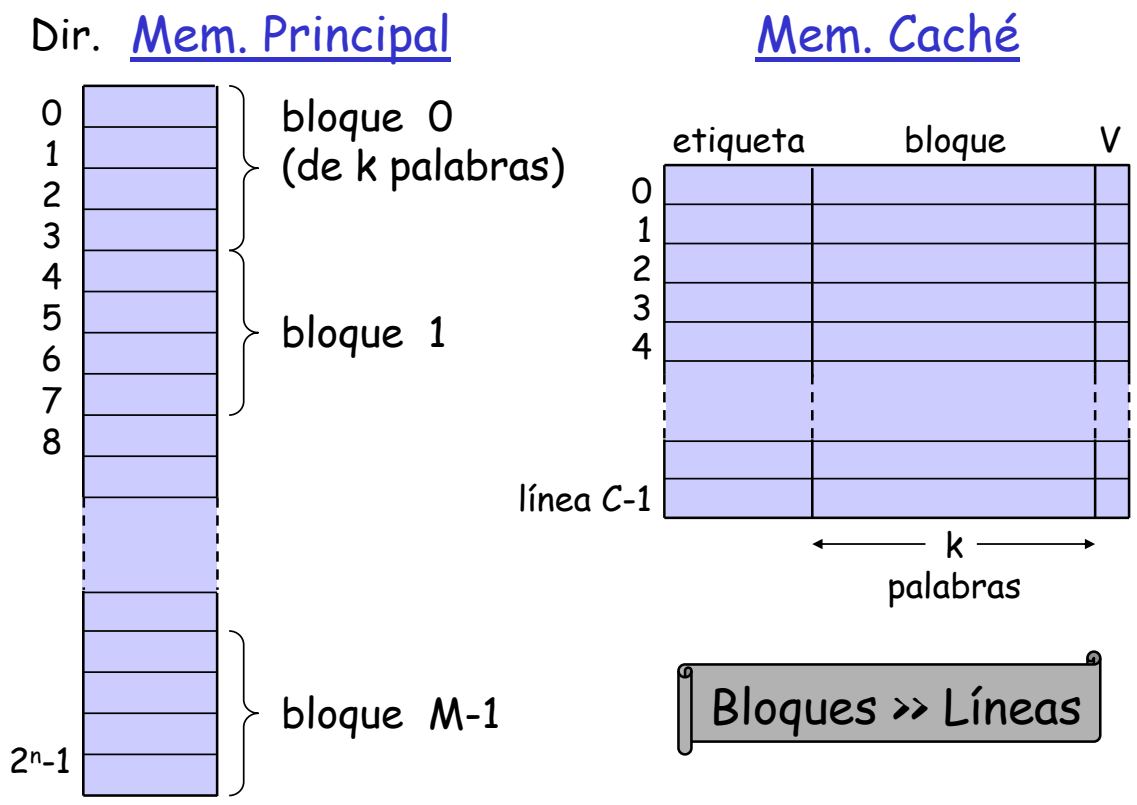
¿Merece la pena la instalación de la memoria caché?

Arriba se muestra la solución.

Para este ejemplo supondremos que cuando se produce una falta de caché, la carga del bloque en la caché se produce al mismo tiempo que se lleva la palabra referenciada a la CPU, es decir, que se realiza con carga directa, con lo que la sobrecarga por el tiempo de carga del bloque resulta nula.

Si dividimos el tiempo medio de acceso sin caché, por el tiempo medio de acceso con caché, se obtiene el índice de mejora, el cual da una idea del orden de magnitud en la mejora del tiempo de acceso.

Estructura

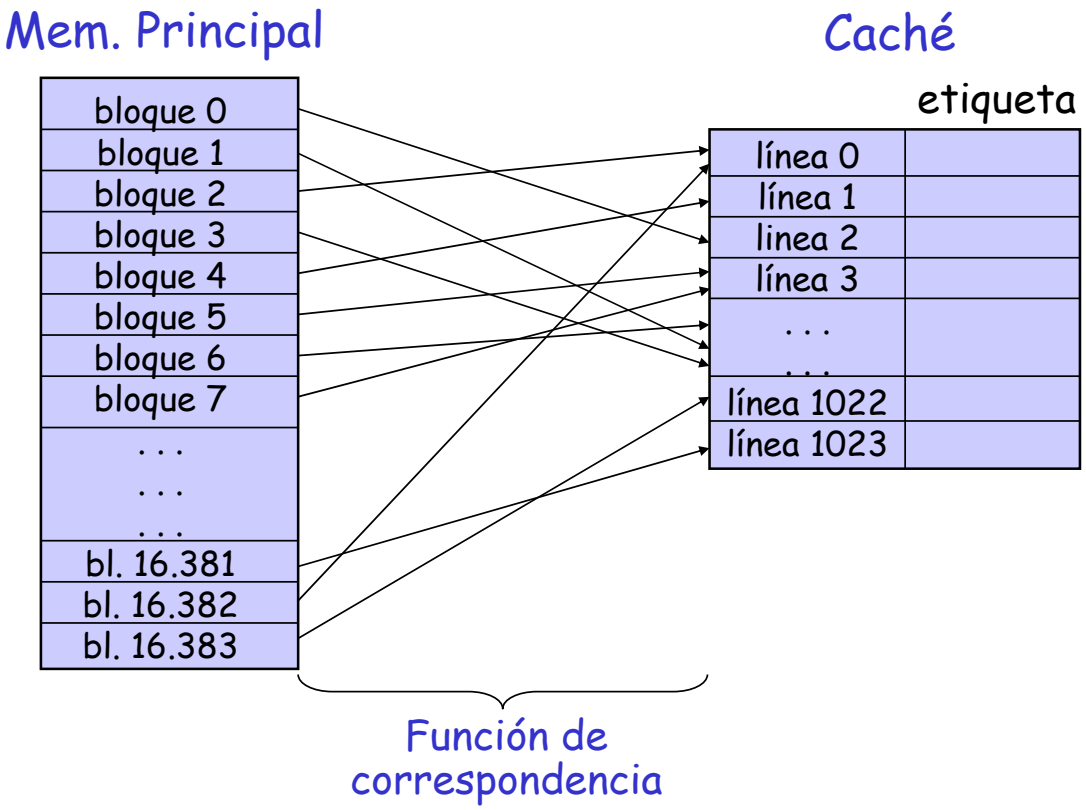


Sabemos que la memoria principal está formada por un máximo de 2^n celdas direccionables, cada una con una única dirección de n bits. Desde el punto de vista de la caché, esta misma memoria principal se considera formada por M bloques de K celdas cada uno; es decir, por $2^n/K$ bloques. La caché por su parte, está formada por C entradas de K celdas cada una, tal que el número de entradas de la caché es mucho menor que el número de bloques de la memoria principal. Cuando se lee una celda de un bloque de memoria, ese bloque se transfiere a la caché, por lo que en un momento dado, la caché está cargada con un subconjunto de los bloques de memoria.

Ya que hay más bloques de memoria que entradas o "líneas" en la caché, cada entrada de la caché no puede estar permanentemente dedicada a un bloque concreto de la memoria principal. Por esto, cada entrada de la caché tiene una **etiqueta** que identifica al bloque que tiene cargado en ese momento. Una etiqueta es algo que diferencia a todos los bloques que pueden estar en una misma línea. Como veremos, esta etiqueta suele estar formada por los bits más significativos de la dirección del bloque en memoria principal..

Cada entrada de la caché también debe disponer de un bit V que indique si esa entrada está **ocupada** por algún bloque (entrada válida) o todavía no se ha cargado con ningún bloque desde que se arrancó la máquina. Este bit se pone a cero en la inicialización de la caché, y se activa cuando se trae un bloque a la línea por primera vez.

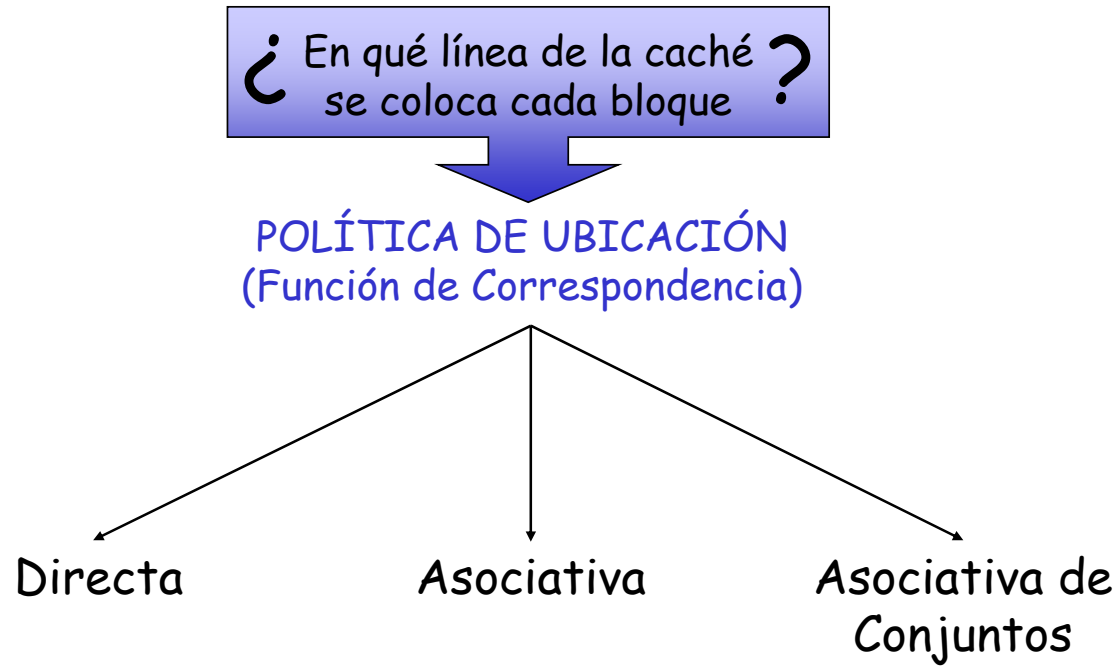
Políticas de Ubicación



Ya que hay menos entradas o líneas de caché que bloques en memoria principal, se nos plantea un problema cuando traemos un bloque de memoria a la caché: ¿Dónde ponemos el bloque? es decir ¿en qué entrada de la caché ponemos el bloque? De igual manera, al buscar un bloque en la caché, debemos saber dónde podría estar ubicado este bloque.

Este problema se resuelve según una **política de ubicación**. Cada política de ubicación utiliza una **función de correspondencia** entre las direcciones de los bloques en memoria principal y sus direcciones en la caché. Así, diremos que las diferentes políticas de ubicación simplemente utilizan distintas funciones de correspondencia.

...Políticas de Ubicación



Las tres funciones de correspondencia que se suelen utilizar son las siguientes:

- Correspondencia Directa
- Correspondencia Asociativa
- Correspondencia Asociativa de Conjuntos

A continuación vamos a tratar con cierto detalle cada una de estas funciones de correspondencia. Para cada una de ellas veremos su estructura general acompañada de un ejemplo, que en los tres casos supondrá lo siguiente:

1. El tamaño de la caché es de 4 Kbytes.
2. Los datos se transfieren entre la memoria principal y la caché en bloques de 4 bytes. Esto quiere decir que la caché está organizada en 1024 líneas de 4 bytes cada una.
3. La memoria principal es de 64 Kbytes, pudiendo direccionar a nivel de byte mediante direcciones de 16 bits. Esto quiere decir que, a efectos de la caché, la podemos considerar como una memoria de 16 Kbloques de 4 bytes cada uno.

Correspondencia Directa

Función de Correspondencia

Num_Línea = NumBloque módulo Líneas_en_caché

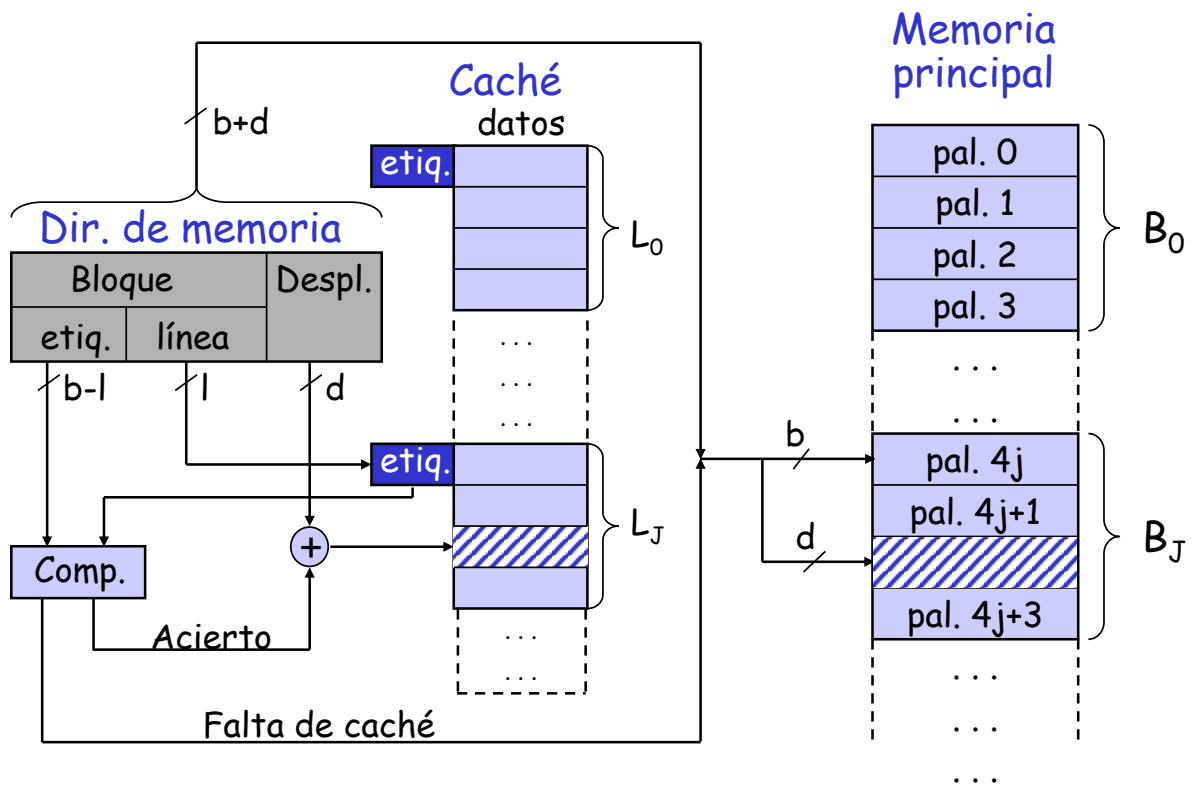
Caché de L líneas

Línea caché	Bloques de memoria principal
0	0 , L , 2L , 3L , ...
1	1 , L+1 , 2L+1 , 3L+1 , ...
. . .	. . .
L-1	L-1 , 2L-1 , 3L-1 , 4L-1 , ...

La función más sencilla de correspondencia es la conocida como **correspondencia directa**, según la cual cada bloque de memoria principal solamente puede ubicarse en una única línea de la caché. La línea que le corresponde a cada bloque se obtiene mediante este algoritmo:

Línea_caché = *Número_de_bloque* módulo *Líneas_en_la caché*

En la transparencia pueden verse los bloques de memoria principal que pueden ubicarse en cada línea de la caché si son referenciados por la CPU.



Este algoritmo se implementa fácilmente a partir de las direcciones que genera la CPU. En lo que concierne a la caché, cada dirección de memoria consta de tres campos:

Desplazamiento: Los *d* bits menos significativos identifican una única celda de memoria dentro de un bloque de memoria principal. Es decir, es el desplazamiento de la celda dentro de su bloque.

Los restantes *b* bits de la dirección indican uno de los 2^b bloques de memoria principal. Como todos los bloques de memoria principal no caben en las *L* líneas de la caché, ésta interpreta estos *b* bits como una composición de dos campos: la línea y la etiqueta.

Línea: Este campo indica la línea en la que debe ubicarse o localizarse un bloque de memoria principal. Está formado por los *l* bits menos significativos de los *b* bits de mayor peso de la dirección, e indica una de las *L* líneas de la caché, pues $2^l = L$.

Pero claro, va a haber muchos bloques a los que les corresponda la misma línea de caché. Concretamente, los restantes *b-l* bits, los de mayor peso de la dirección, indican a cuántos bloques les corresponde la misma línea en la caché.

En la figura de la diapositiva anterior se puede ver que en una caché con *L* líneas, a la línea *0* le corresponden los bloques número *0, L, 2L, 3L, ...*; a la línea *1* le corresponden los bloques *1, L+1, 2L+1, 3L+1, ...*; y a la línea *L-1* le corresponden los bloques *L-1, 2L-1, 3L-1, ...*

Etiqueta: Lo que va a diferenciar a todos los bloques a los que les corresponda la misma línea son los *b-l* bits de mayor peso, esto es, el campo de "etiqueta". Obsérvese que aunque una línea de caché puede corresponder a varios bloques, todos esos bloques tendrán una etiqueta distinta.

Cuando la CPU realiza una lectura, la dirección se divide en estos tres campos. Tomando los *b* bits del campo de bloque se obtiene el número de bloque de memoria principal. Con el algoritmo arriba indicado se obtiene la línea que le corresponde al bloque. Si la línea no está ocupada, se trae el bloque desde memoria principal a esa línea, y con el desplazamiento que indican los *d* bits de menor peso de la dirección se obtiene la celda dentro del bloque. Los *b-l* bits de mayor peso de la dirección deben ponerse en el campo de etiqueta de la línea correspondiente de la caché.

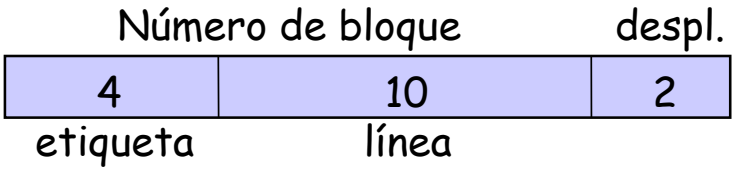
En una referencia posterior, cuando se compruebe si el bloque referenciado está en la caché, si la entrada correspondiente está ocupada, hay que comprobar si el bloque de esa entrada es el que corresponde a la dirección que se está referenciando. Para ello simplemente hay que comprobar que el campo de etiqueta de la dirección es igual a la etiqueta de la línea que corresponde a ese bloque. Si no es así, habrá que traer el bloque de memoria y sustituir al que estaba en esa línea. Esto puede representar un problema.

Ejemplo

Tamaño caché: 4 Kbytes
Tamaño bloque: 4 bytes
→ Caché de 1 Klíneas de 4 bytes

Memoria principal: 64 Kbytes (16 Kbloques de 4 bytes)

Dirección M.P.



Línea caché	Bloques de memoria principal
0	0, 400, 800, C00, ..., 3C00
1	1, 401, 801, C01, ..., 3C01
. . .	. . .
3FF	3FF, 7FF, BFF, ..., 3FFF

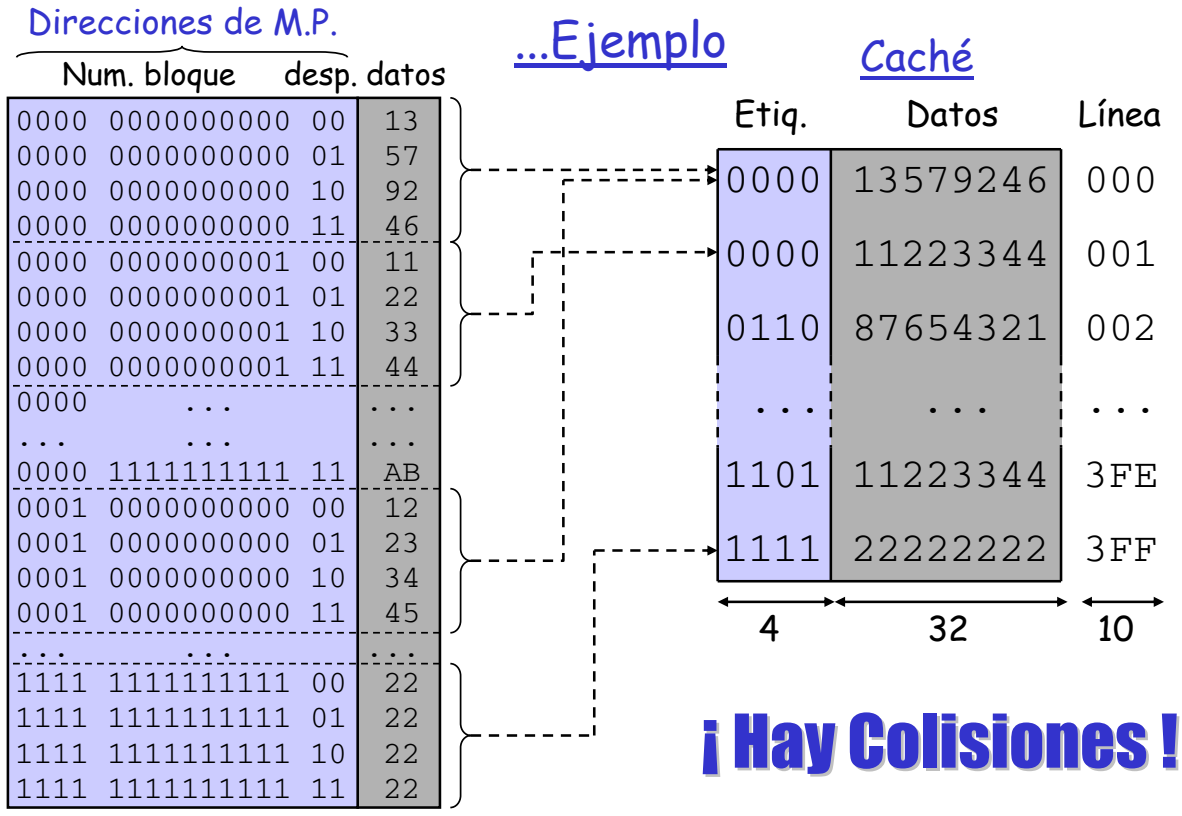
Veamos un ejemplo de ubicación mediante correspondencia directa. Para ello, supondremos los siguientes valores:

1. El tamaño de la caché es de 4 Kbytes.
2. Los datos se transfieren entre la memoria principal y la caché en bloques de 4 bytes. Esto quiere decir que la caché está organizada en 1024 líneas de 4 bytes cada una.
3. La memoria principal es de 64 Kbytes, pudiendo direccionar a nivel de byte mediante direcciones de 16 bits. Esto quiere decir que, a efectos de la caché, la podemos considerar como una memoria de 16 Kbloques de 4 bytes cada uno.

Veamos el formato de las direcciones. Los 16 Kbloques de la memoria principal se referencian mediante los 14 bits de más peso de la dirección; los dos bits de menor peso constituyen el desplazamiento de la palabra dentro del bloque. En la caché, por su parte, el número de línea se expresa mediante 10 bits. La ejecución del algoritmo de la función de correspondencia para averiguar la línea que le corresponde a un bloque de memoria (dividir un número de 14 bits entre otro de 10 y tomar el resto) es lo mismo que tomar directamente los 10 bits de menor peso del dividendo, es decir, del número de bloque.

En la parte inferior de la transparencia podemos ver las correspondencias entre bloques y líneas para este ejemplo.

...Correspondencia Directa



Las direcciones de memoria están descompuestas en número de bloque y desplazamiento. El número de bloque lo mostramos dividido en dos campos, para ver claramente la etiqueta de cada bloque y la línea que le corresponde.

Así, podemos ver cómo al bloque 0 le corresponde la línea 0; al bloque 1, la línea 1; y en último lugar, al bloque 3FFF le corresponde la última línea, la 3FF.

Como ya sabíamos, al bloque 400 H, también le corresponde la línea 0, pero como se puede apreciar, el contenido de esta línea es el del bloque 0, y es razonable, puesto que la etiqueta de la línea 0 es la correspondiente al bloque 0, y no la del bloque 400 H, que tiene una etiqueta 0001.

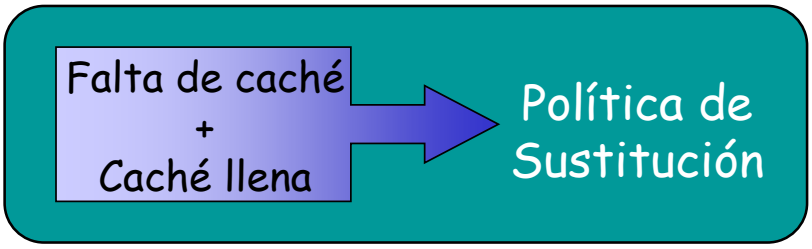
La técnica de la correspondencia directa es simple y económica, pues la ubicación se obtiene directamente a partir del algoritmo de la función de correspondencia.

Sin embargo, el problema que presenta la correspondencia directa son las **colisiones**; es decir, que a cada línea de caché (donde sólo cabe un bloque) le corresponden muchos bloques de memoria principal. Así, si un programa referencia repetidamente dos palabras a cuyos bloques les corresponde la misma línea de la caché, los dos bloques estarán continuamente expulsándose de la caché, con lo que la tasa de aciertos descenderá drásticamente aún cuando la caché no esté completamente ocupada.

Correspondencia Asociativa

Función de Correspondencia

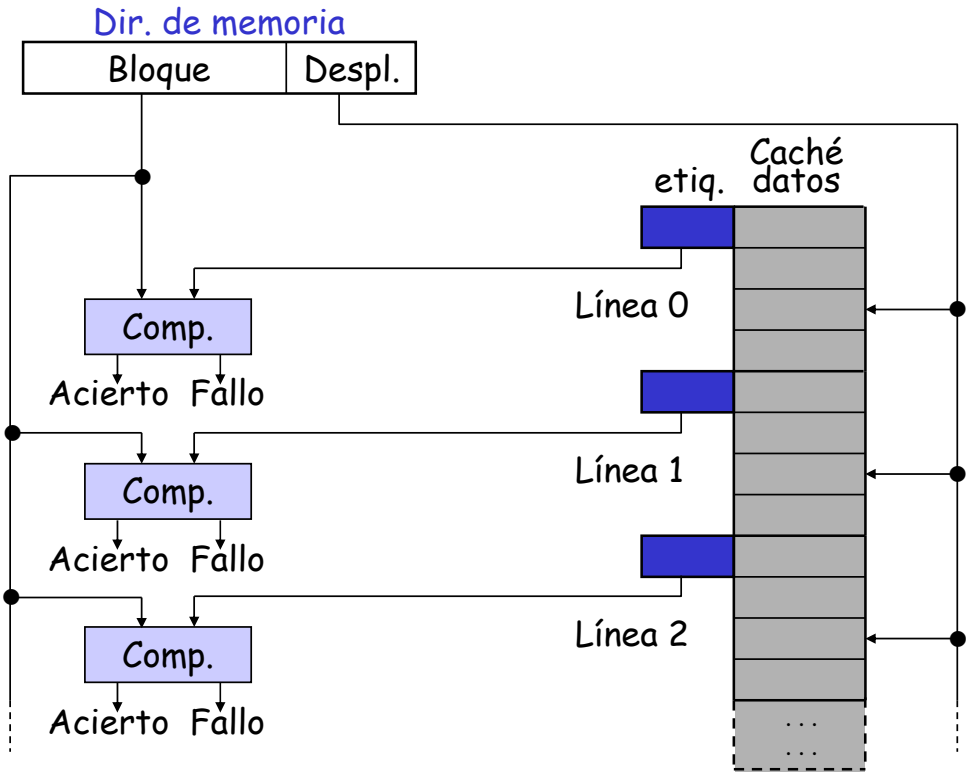
El bloque puede ubicarse en cualquier línea de la caché



Visto el problema de las colisiones que presenta la correspondencia directa, veamos otras alternativas

Con la **correspondencia asociativa** (o completamente asociativa) se solventan los problemas de la correspondencia directa, pues aquí se permite que cada bloque de memoria pueda estar en cualquier línea de la caché, por lo que mientras la memoria caché no esté llena, no habrá que hacer ninguna sustitución. Cuando esté llena y haya que traer un nuevo bloque, habrá que sustituir alguno de los bloques según la política de sustitución más apropiada, es decir, la que genere menos faltas de caché.

...Correspondencia Asociativa



Con la correspondencia asociativa, la caché ve cada dirección de memoria formada solamente por dos campos: el desplazamiento dentro del bloque (los bits menos significativos) y el número de bloque o etiqueta (los más significativos). Ahora cada bloque de memoria principal tiene una única etiqueta posible, que es precisamente el número de bloque.

Así, para saber si un bloque está en la caché, su lógica de control debe comparar la etiqueta de la dirección generada por la CPU con todas las etiquetas de la caché. Para que estas comparaciones puedan realizarse rápidamente, cada entrada de la caché cuenta con un comparador, de tal manera que las comparaciones de la etiqueta de la dirección de memoria con todas las etiquetas de las líneas de la caché se realizan en paralelo. (Este tipo de memorias se denominan **memorias asociativas**).

Con este esquema hay flexibilidad para ubicar un bloque en cualquier línea de la caché, y su espacio se puede aprovechar más eficientemente, pues cuando se trae un bloque a la caché nunca habrá que reemplazar a ninguno de los que ya estaban cargados a menos que todas las líneas estén ocupadas. Con esta técnica ya no deben producirse las repetidas expulsiones mutuas de dos bloques que teníamos con la correspondencia directa. Los algoritmos de sustitución que veremos más adelante se diseñarán precisamente para mejorar lo más posible la tasa de aciertos.

La desventaja obvia de la correspondencia asociativa es el incremento económico que genera la electrónica adicional necesaria.

...Correspondencia Asociativa

Nuestro Ejemplo

Tamaño caché: 4 Kbytes
Tamaño bloque: 4 bytes
→ Caché de 1 Klínea de 4 bytes

Memoria principal: 64 Kbytes (16 Kbloques de 4 bytes)

Dirección M.P.

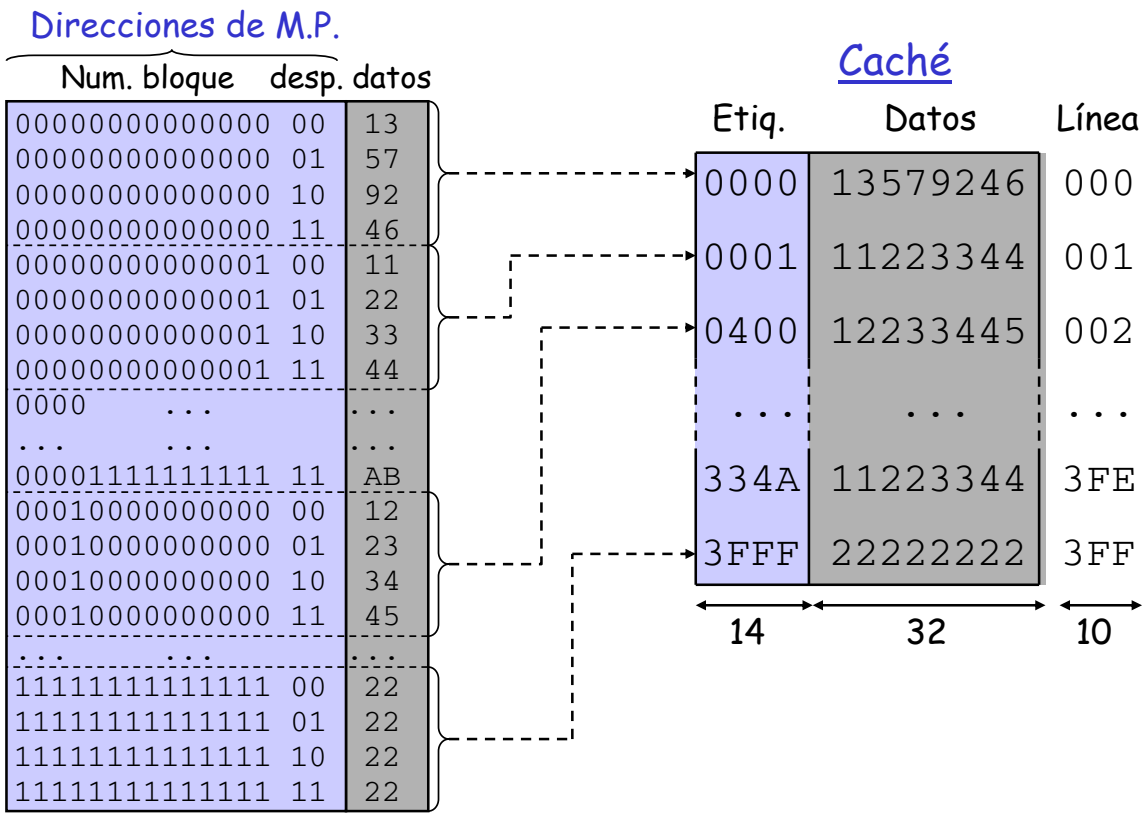
Número de bloque (etiqueta)	despl.
14	2

Línea caché	Bloques de memoria principal
0	cualquiera
1	cualquiera
. . .	. . .
3FF	cualquiera

En esta transparencia mostramos la aplicación de la función de correspondencia asociativa a la arquitectura del ejemplo que estamos utilizando para ilustrar las políticas de ubicación.

Como vemos, la dirección de 16 bits ahora solamente se descompone en dos campos: el desplazamiento, que tiene que seguir siendo de 2 bits, pues lo impone el tamaño del bloque; y el número de bloque, de 14 bits, que se utiliza como etiqueta.

...Correspondencia Asociativa



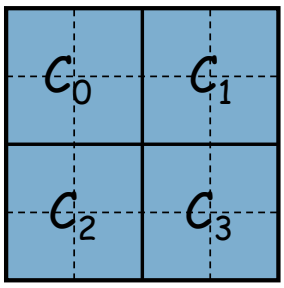
Como hemos dicho, la dirección de 16 bits ahora solamente se descompone en dos campos: el desplazamiento, que tiene que seguir siendo de 2 bits, pues lo impone el tamaño del bloque; y el número de bloque, de 14 bits, que se utiliza como etiqueta.

Esto quiere decir que en cada línea de la caché, junto con cada bloque de 4 bytes deben almacenarse también los 14 bits de su etiqueta correspondiente.

Ya que esta correspondencia no fuerza ninguna ubicación concreta para cada bloque de memoria principal, ya no hay colisiones como en la correspondencia directa, sino que al traer un bloque a la caché, si ésta está totalmente ocupada, simplemente debe sustituirse alguno de los bloques que ya estaban por el recién traído. Puesto que no se fija de antemano cuál debe ser el bloque a sustituir, no se corre el peligro de la correspondencia directa, en la que podía darse el caso de tener que sustituir un bloque que se estaba referenciando muy a menudo. La política de sustitución se encargará de elegir el bloque que al ser sustituido genere el menor perjuicio posible a la tasa de aciertos de la caché.

Como se muestra en el ejemplo, con esta correspondencia se consigue que, normalmente, los últimos bloques referenciados por la CPU se encuentren en la caché, sin que uno de ellos haya tenido que expulsar a otro recientemente referenciado para cargarse en la caché.

Correspondencia Asociativa de Conjuntos



La caché se divide en *C* conjuntos de *L* líneas cada uno

Función de Correspondencia

Num_Conjunto = Num_bloque módulo Num_Conjuntos

El bloque se ubica en cualquier línea del conjunto

A muchos bloques les corresponderá el mismo conjunto

Política de sustitución dentro de cada conjunto

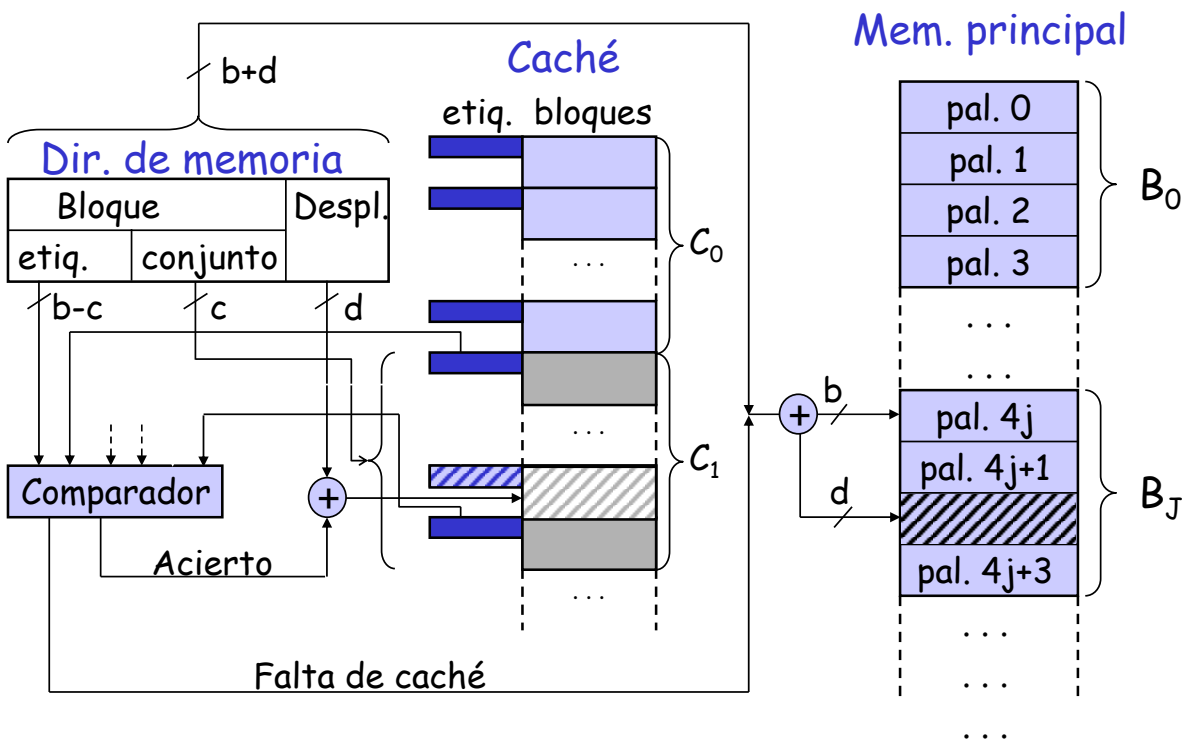
La **correspondencia asociativa de conjuntos** combina la economía de la correspondencia directa con la tasa de aciertos de la correspondencia asociativa. Consiste en agrupar las líneas de la caché en conjuntos, de tal forma que la función de correspondencia permita que un bloque de la memoria principal pueda ubicarse en cualquier línea de un conjunto concreto.

Con esta política, la memoria caché se divide en *C* conjuntos de *L* líneas cada uno. Así, el número *M* de líneas de la caché es $M = C \times L$.

De forma similar a la correspondencia directa, para averiguar el conjunto *c* de la caché que le corresponde a un cierto bloque *b* de memoria principal, se aplica la siguiente correspondencia:

$$c = b \text{ módulo } C$$

Una vez averiguado el conjunto *c* de la caché que le corresponde al bloque *b*, éste puede ubicarse en cualquiera de las líneas del conjunto *c*.



También de manera equivalente a la correspondencia directa, cuando la CPU suministra una dirección para acceder a la memoria principal, la dirección se descompone en dos campos: *bloque* y *desplazamiento*. Ya que todos los bloques de memoria principal no caben en la caché, el campo *bloque* se divide, a su vez, en otros dos campos: *conjunto* y *etiqueta*. Ya sabemos que la operación *número módulo* 2^n es lo mismo que tomar los *n* bits de menor peso del *número*, por lo que el conjunto asignado a un bloque de memoria se obtiene con los *c* bits siguientes al desplazamiento. Por último, los bits restantes (los de mayor peso) forman la etiqueta.

Un bloque de memoria solamente puede estar en un conjunto de la caché, pero dentro del conjunto hay varios bloques, por ésto es necesario disponer de la etiqueta, pues no puede haber dos bloques del mismo conjunto con la misma etiqueta.

Obsérvese que si el número de líneas por conjunto se lleva a los **casos extremos**, se da lugar a las otras dos correspondencias que hemos visto. Cuando el número de líneas por conjunto es 1, se está en el caso de la correspondencia directa; mientras que si la caché está formada por un único conjunto que contiene todas las líneas de la caché, se trata de una correspondencia completamente asociativa.

Normalmente **se suelen utilizar 2 líneas por conjunto** (memoria asociativa de dos vías), lo cual mejora notablemente la tasa de aciertos de la correspondencia directa. A medida que se aumenta el número de líneas por conjunto, aumenta el coste de la memoria pero sin conseguir una mejora significativa.

Con esta técnica se disminuye mucho el coste de la electrónica adicional de la correspondencia asociativa, pues mientras que en ésta se requieren tantos comparadores como líneas, en la asociativa de conjunto de dos vías, solamente son necesarios dos comparadores.

...Correspondencia Asociativa de Conjuntos

Nuestro Ejemplo

Tamaño caché: 4 Kbytes
Tamaño bloque: 4 bytes
→ Caché de 1 Klínea de 4 bytes

Conjuntos de 4 bloques → Caché de 256 conjuntos

Memoria principal: 64 Kbytes (16 Kbloques de 4 bytes)

Dirección M.P.

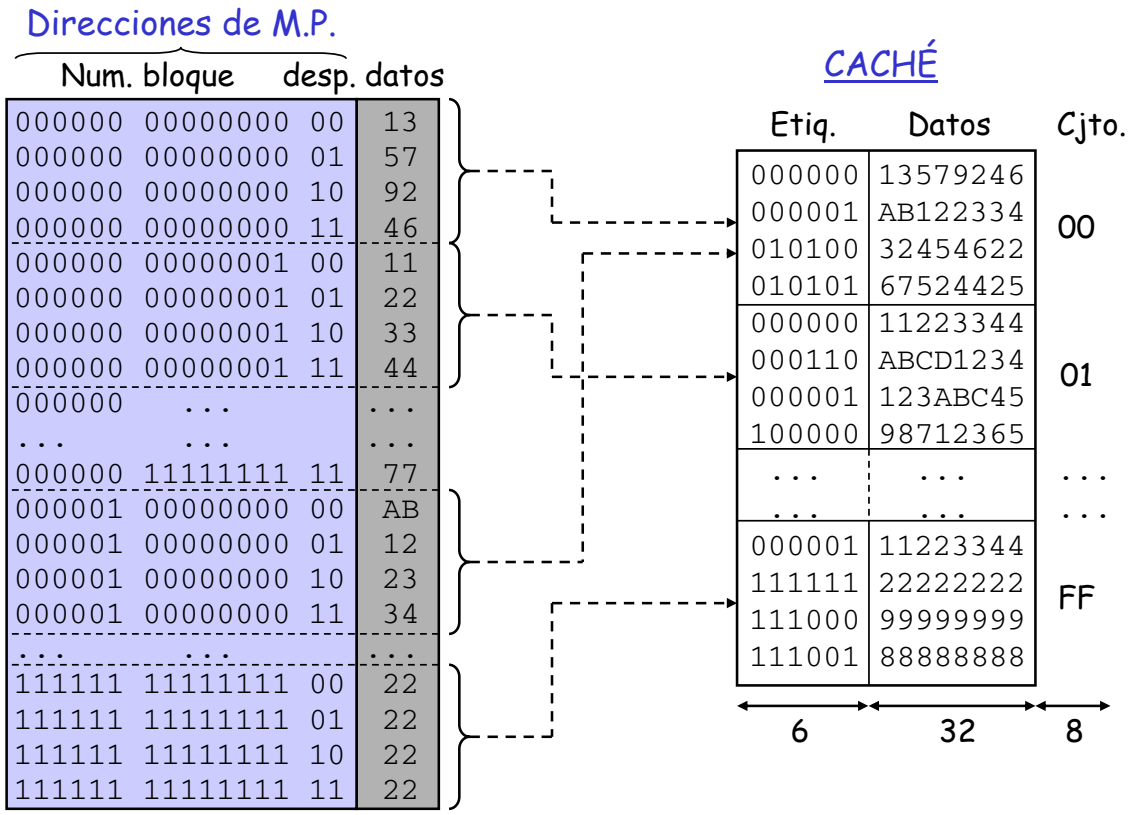
Número de bloque		despl.
6	8	2
etiqueta	conjunto	

Cjto. caché	Bloques de memoria principal
0	0, 100, 200, ..., 3F00
1	1, 101, 201, ..., 3F01
. . .	. . .
FF	FF, 1FF, 2FF, ..., 3FFF

Veamos ahora cómo aplicar la correspondencia asociativa de conjuntos a nuestro ejemplo tipo.

Ahora la dirección de 16 bits vuelve a descomponerse en tres campos. El desplazamiento sigue estando indicado por los dos bits de menor peso, puesto que los bloques siguen siendo de 4 palabras. En cuanto al número de bloque, queda indicado por los 14 bits de mayor peso, aunque en este caso, para conocer la ubicación de cada bloque en la caché solamente necesitamos los 8 bits de menor peso del número de bloque, que es el resultado de la operación **`Num_Bloque módulo Num_Conjuntos`**. Así, tenemos que los bloques con número 0, 100H, 200H, etc., deben ubicarse en el conjunto 0 de la caché; los que tienen número 1, 101H, 201H, etc., corresponden al conjunto 1, y así sucesivamente.

...Correspondencia Asociativa de Conjuntos



Al igual que en la correspondencia directa, nos encontramos con que hay colisiones, es decir, hay muchos bloques que les corresponde el mismo conjunto. Sin embargo, mientras que en la correspondencia directa solamente podía haber un bloque en una dirección, ahora tenemos que en una misma dirección de la caché puede haber tantos bloques como permita el tamaño del conjunto. En nuestro ejemplo los conjuntos son de 4 bloques, como el Motorola 68040 y el PowerPC 604, que también utilizan una memoria asociativa con conjuntos de 4 bloques (memoria asociativa de conjuntos de 4 vías). Los Pentium, sin embargo, disponen de una caché asociativa de conjuntos de 2 vías.

Volviendo a nuestro ejemplo, podemos ver que al primer bloque de memoria principal (el bloque 0) le corresponde el conjunto 0 de la caché, y al segundo bloque, el conjunto 1. Como ya sabíamos, nos encontramos con que al bloque 100H también le corresponde el conjunto 0, pero en esta ocasión, para cargar el bloque 100H no hay que expulsar al bloque 0 (como sucedía en la correspondencia directa), ya que ahora en cada conjunto se pueden ubicar hasta 4 de los bloques a los que la función de correspondencia otorga la misma dirección. Para diferenciar los distintos bloques que corresponden al mismo conjunto se utiliza la etiqueta de 6 bits que tiene cada bloque. No puede haber dos bloques con la misma etiqueta que les corresponda el mismo conjunto.

De esta manera se evita en gran medida el problema de la correspondencia directa, en la que dos bloques muy usados en un bucle podían estar expulsándose mútuamente, desaprovechando así la eficacia de la memoria caché.

Políticas de Sustitución

Cuando se produce una falta de caché y hay que traer el bloque desde memoria principal, si no hay una línea libre para el bloque, habrá que sustituir alguno de los que están en la caché por el recién referenciado.

Si se utiliza correspondencia directa, no hay ninguna elección, hay que sustituir el bloque de la única línea en la que se puede ubicar el bloque referenciado. Si se trata de correspondencia asociativa de conjuntos, se puede elegir entre cualquiera de los bloques del conjunto que le corresponde al bloque referenciado. Y si la función de correspondencia es la completamente asociativa, se puede elegir para sustituir cualquiera de los bloques que están en la caché. Así, tenemos que en los dos últimos tipos de correspondencias necesitamos una **política de sustitución**. Esta cuestión es extremadamente importante, pues la política utilizada es un factor determinante para la tasa de aciertos del sistema.

Hay dos enfoques a la hora de elegir una política de sustitución: el que tiene en cuenta la estadística de utilización de los bloques (la historia de uso), y el que no lo tiene en cuenta. Entre los primeros se encuentran las políticas LRU y LFU, y como representantes del segundo enfoque está la política *random* y la FIFO.

LRU (*Least Recently Used*). En general, el objetivo es mantener en la caché los bloques que tienen más probabilidades de ser accedidos en un futuro próximo, pero no resulta nada fácil saber esto. No obstante, el principio de localidad nos dice que en ciertas áreas de memoria, y durante un periodo razonable de tiempo, hay una alta probabilidad de que los bloques que acaban de ser referenciados recientemente sean referenciados otra vez en un futuro próximo. Por eso, cuando hay que seleccionar un bloque víctima, parece razonable elegir el que lleva más tiempo sin ser referenciado (el menos recientemente referenciado, *the least recently used*).

Para implementar este sistema se requiere añadir un contador a cada línea de la caché, de tal manera que cada vez que se referencia un bloque su contador se pone a cero, y los de los demás (del conjunto o de toda la memoria) se incrementan en uno. Cuando se trae un nuevo bloque, se elige como víctima el que tiene el contador más alto, y al nuevo bloque se le pone el contador a cero.

Esta política de sustitución falla cuando se está accediendo de forma secuencial y cíclica a elementos de una matriz que no cabe completamente en la caché.

Una política parecida es la **LFU (*Least Frequently Used*)**, en la cual se sustituye el bloque menos referenciado. Ésta también se implementa con ayuda de contadores.

Como políticas no basadas en la estadística de uso, tenemos la **FIFO (*First In, First Out*)** que ofrece unos resultados ligeramente inferiores a la LRU, y que se implementa con ayuda de un *buffer* circular. Los problemas que presenta se deben al hecho de que solamente tiene en cuenta el tiempo que lleva un bloque en la caché, y no cuáles han sido las últimas referencias; por esto también puede fallar con algunas matrices.

La política ***Random***, que elige al azar el bloque a sustituir, ofrece una buena tasa de aciertos, no tiene el problema de la LRU con las matrices, y es fácil y económico de implementar.

	Memoria Asociativa de Conjuntos					
	Dos Vías		Cuatro Vías		Ocho Vías	
Tamaño	LRU	*Random*	LRU	*Random*	LRU	*Random*
16 Kb	5,18%	5,69%	4,67%	5,29%	4,39%	4,96%
64 Kb	1,88%	2,01%	1,54%	1,66%	1,39%	1,53%
256 Kb	1,15%	1,17%	1,13%	1,13%	1,12%	1,12%

Comparación de tasa de fallos según diversos tamaños y políticas de sustitución

Las estadísticas dicen que la política de ubicación influye mucho más en las prestaciones de la caché que la política de sustitución. A su vez, no hay una política de sustitución claramente mejor que otra, pues dependen de la política de ubicación utilizada y del tamaño de la caché.

En la tabla de arriba podemos observar el porcentaje de fallos de caché con distintos tamaños de caché y diversas configuraciones de número de vías y políticas de sustitución.

Merece la pena comentar la drástica caída al pasar de una caché de 16 Kbytes a una de 64; bajando muy poco al aumentarla hasta 256 Kbytes. Esto es razonable, ya que cuando en la caché cabe el conjunto de trabajo que se está utilizando (conjunto de bloques muy accedidos durante la ejecución de la porción de un programa), no se consigue nada por aumentar el tamaño de la caché.

También se puede apreciar la poca diferencia de tasa de fallos que hay entre las políticas LRU y “al azar” o *Random*.

Parece que aumentando el número de vías disminuye ligeramente la tasa de fallos. No obstante, las memorias cachés sueles ser de 2 o 4 vías.

Políticas de Escritura

Cuestiones ante una Operación de Escritura

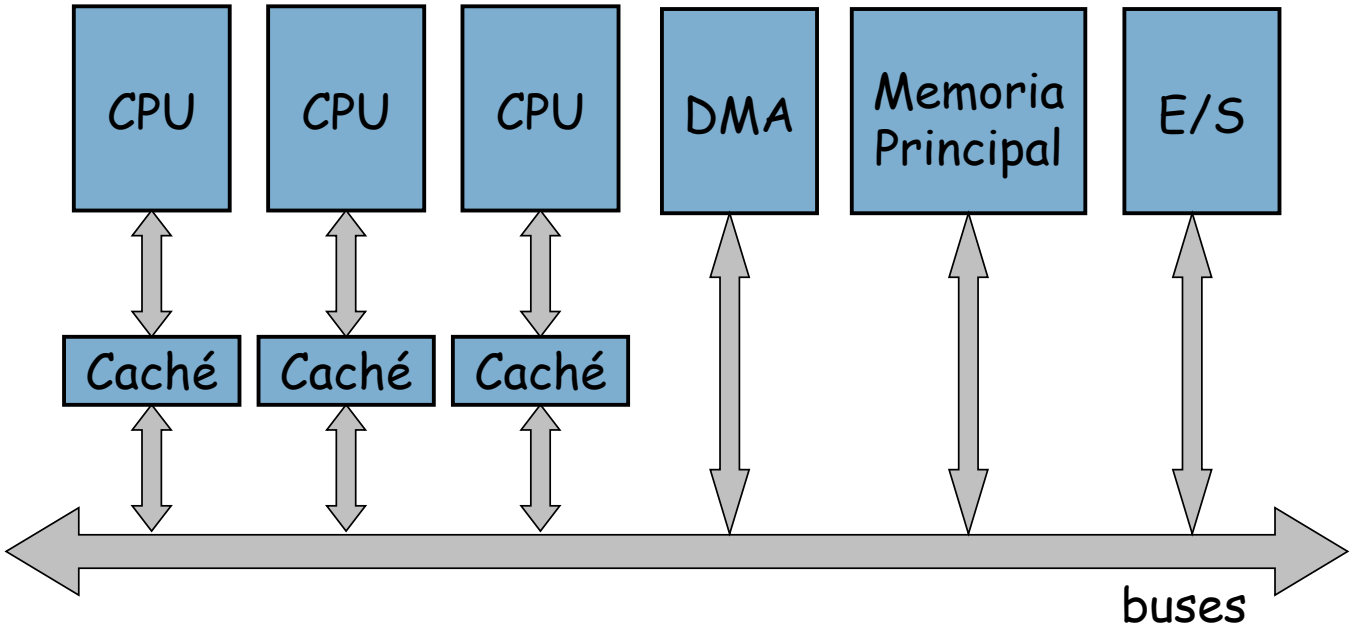
- Problema de la Coherencia de las Cachés
- Falta de Caché en Escritura

Hay dos cuestiones que tratar en relación con la operación de escritura en un sistema con memoria caché:

- Problema de la Coherencia de Cachés
- Falta de Caché en Escritura

Veámoslas en las siguientes páginas.

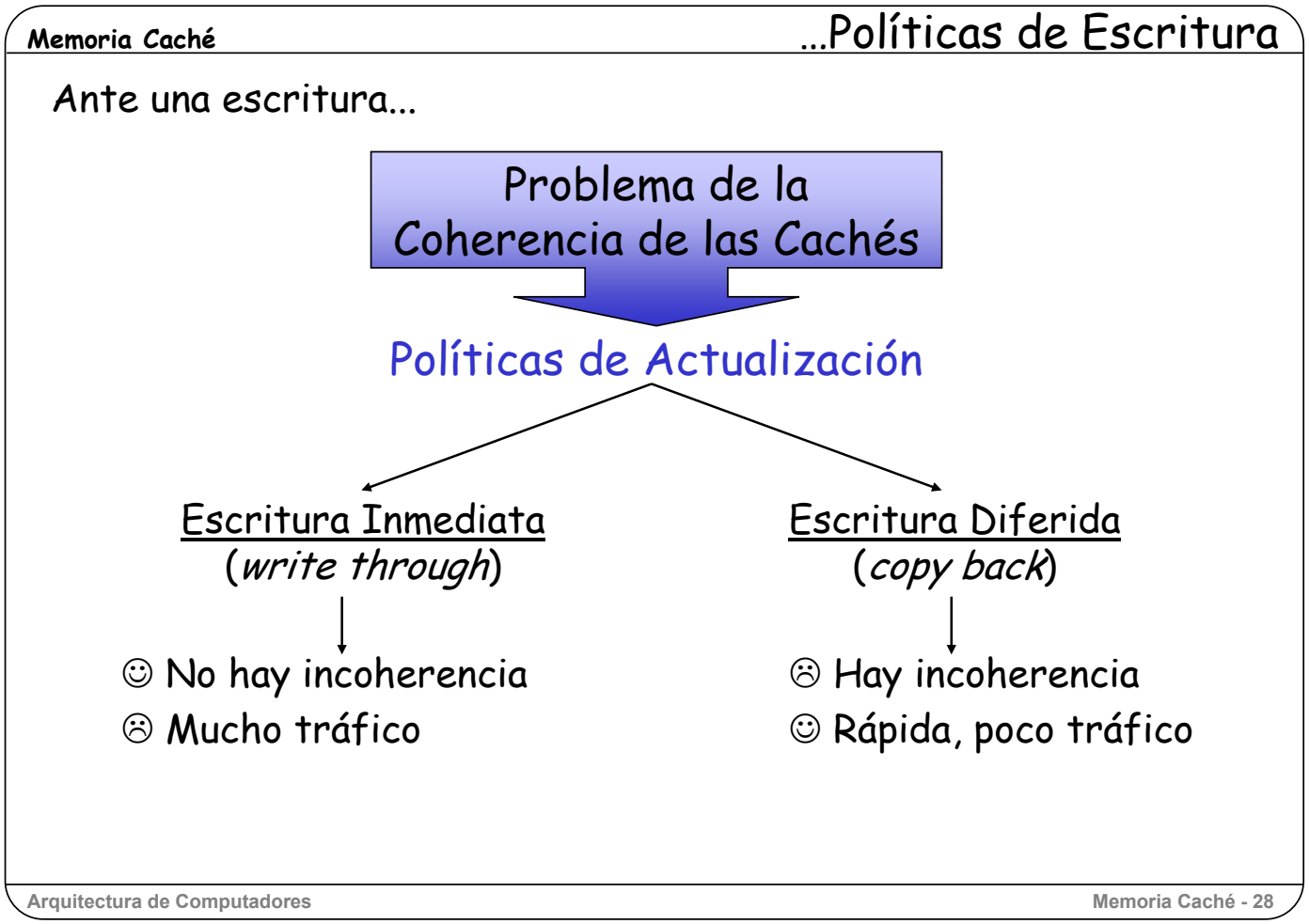
El Problema de Coherencia de las Cachés



El problema de la coherencia de cachés tiene que ver con la **política de actualización**.

Aquí hay dos situaciones a considerar. En un ordenador puede haber más de un elemento que acceda a la memoria principal, pues en un ordenador actual no es raro encontrarnos con varias CPU's o, simplemente, los dispositivos de entrada/salida gobernados directamente desde alguna CPU o a través del controlador de DMA. Según esto, cuando desde una CPU se modifica un dato en su caché, el correspondiente dato en memoria principal queda obsoleto, con lo que si desde otra CPU o dispositivo se accede al dato original en memoria principal, resulta que se accede a un dato que no está actualizado. También puede suceder que sea un dispositivo de entrada el que modifica el dato en memoria principal con un nuevo valor, con lo que entonces es el dato de la caché el que queda obsoleto.

Esto es lo que se conoce como el **problema de la coherencia de las cachés**.



El problema de coherencia de las cachés depende de la **política de escritura o actualización** que se utilice.

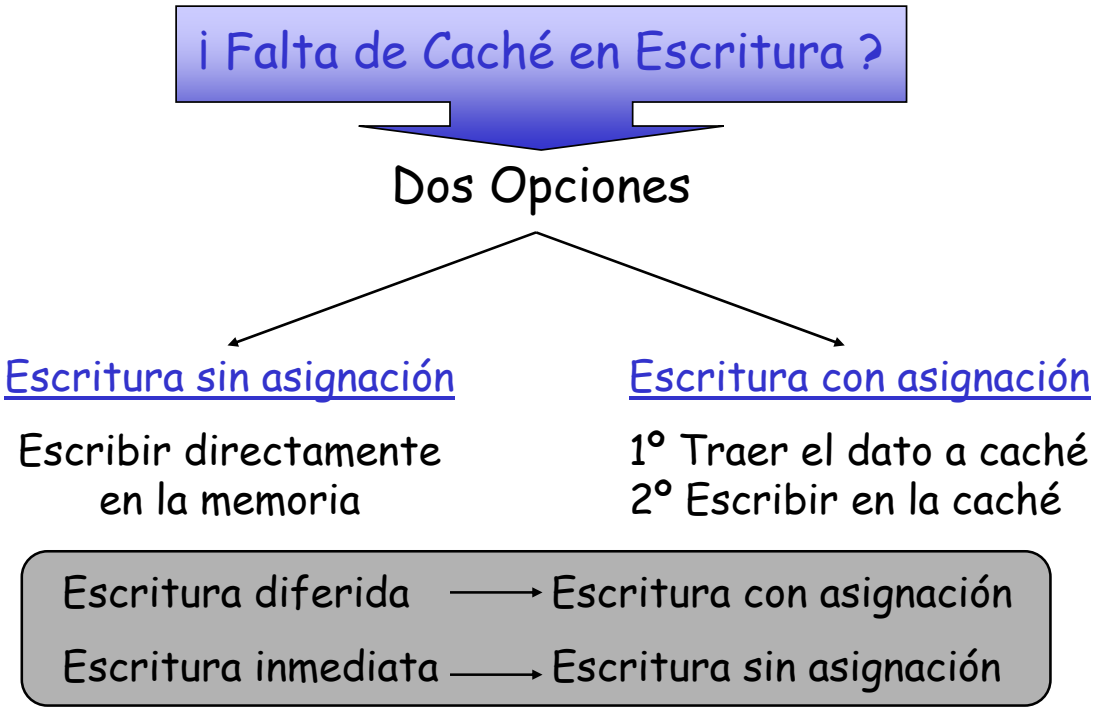
La técnica más simple es la **escritura o copia inmediata** (*write through*), según la cual todas las escrituras se realizan tanto en la caché como en la memoria principal, asegurando así que la memoria principal siempre está actualizada, lo cual simplifica el problema de la coherencia en entornos multiprocesadores. El inconveniente que presenta es el tráfico que genera en los buses del sistema. Podría pensarse que también supone una sobrecarga de tiempo, puesto que además de escribir en la caché, hay que escribir también en la memoria principal, pero está última operación puede realizarse en paralelo con la escritura en memoria caché o con otras actividades por lo que, en la práctica, no supone un tiempo adicional.

La otra alternativa es la **escritura o copia diferida** (*copy back*), en la cual una operación de escritura solamente escribe en la caché.

En esta última alternativa, cuando se escribe o actualiza un bloque de la caché, se activa el bit "modificado" (*dirty bit*) asociado a esa línea, de tal manera que cuando hay que reemplazar un bloque, si su bit "modificado" está activado entonces el bloque debe actualizarse en memoria principal, por lo que hay que escribirlo en ella antes de ocupar la línea de la caché con el nuevo bloque. El problema que tiene es que, en un momento dado, hay datos en memoria principal que no están actualizados, lo cual puede originar problemas de coherencia de datos.

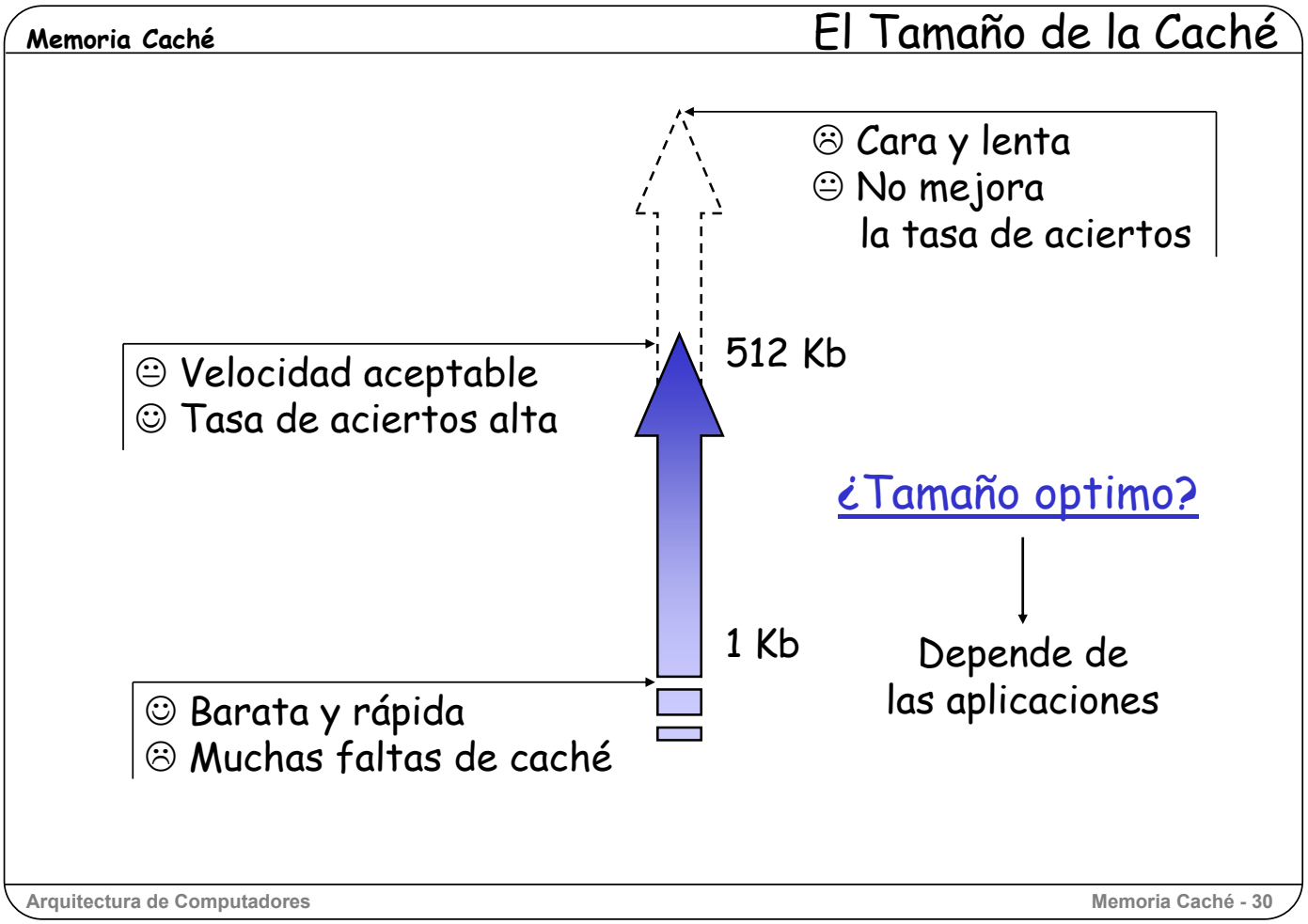
...Políticas de Escritura

Ante una escritura...



La otra pregunta es ¿qué hacer cuando se produce una **falta de caché en escritura**? La cuestión concreta es si ante una falta de caché en escritura, se debe escribir directamente en la memoria principal (escritura sin asignación de caché), o se debe traer el dato a la caché y escribir en la caché (escritura con asignación).

Aunque cualquiera de estas opciones se puede utilizar con las dos políticas de actualización, la escritura diferida suele utilizar la escritura con asignación, esperando así que las subsiguientes escrituras a ese bloque se realicen directamente en la caché. Las escrituras inmediatas suelen ser sin asignación, ya que las siguientes escrituras a ese bloque tendrán que realizarse forzosamente sobre memoria principal.



Además de las cuestiones generales que hemos tratado, hay otras consideraciones que se deben tener en cuenta en el diseño de una memoria caché: el tamaño total de la memoria caché y el tamaño de cada bloque.

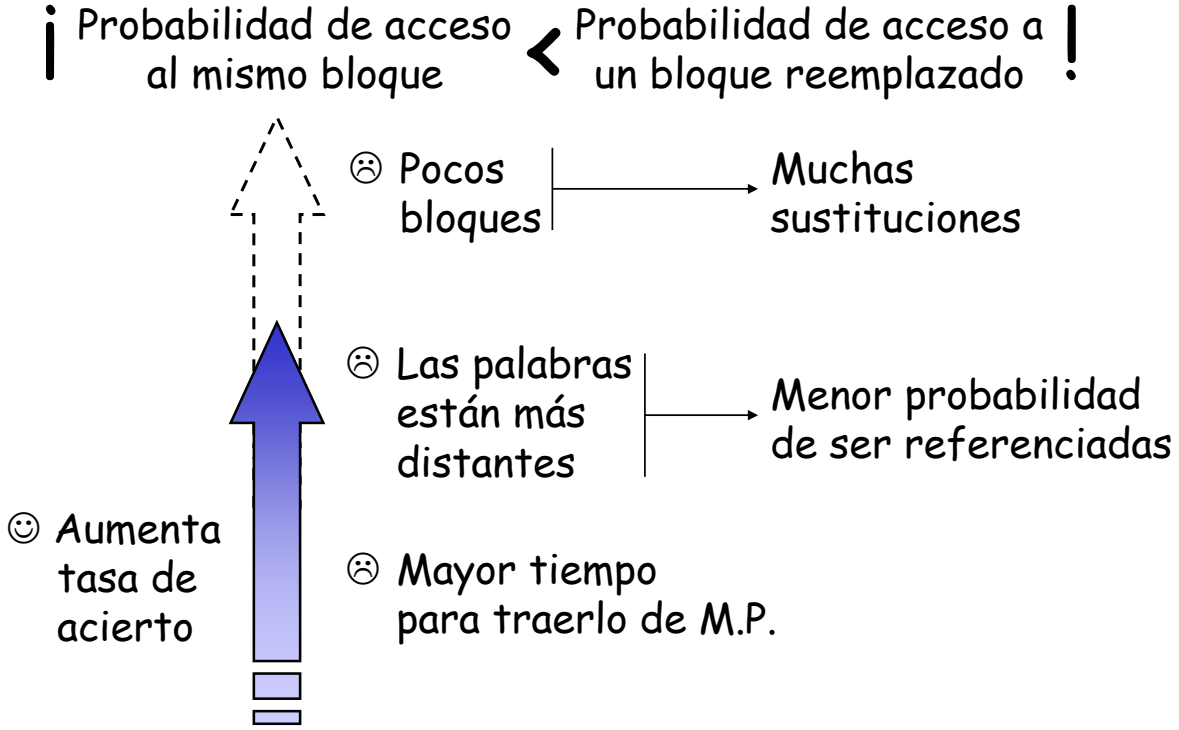
Tamaño de la caché. Ya hemos comentado que nos gustaría que el tamaño de la caché fuera lo suficientemente pequeño como para que el coste medio por bit de memoria total (caché más memoria principal) fuera muy cercano al de la memoria principal, y lo suficientemente grande como para que el tiempo medio de acceso a memoria fuera casi el mismo que el tiempo de acceso a la caché.

Cuanto más grande sea la caché, mayor es el número de puertas necesarias para su direccionamiento, por lo que las grandes cachés tienden a ser ligeramente más lentas que las pequeñas.

Por el principio de localidad temporal, a mayor número de líneas de caché, mayor probabilidad de acierto. No obstante, un programa solamente direcciona unas cuantas áreas de memoria durante un cierto intervalo de tiempo, por lo que a partir de un cierto número de líneas, no se mejorará la tasa de aciertos.

Las prestaciones de la caché dependen mucho de las características del programa en ejecución, por lo que no se puede establecer un tamaño óptimo, aunque algunos estudios estadísticos sugieren que el tamaño óptimo de la caché debe estar entre 1 y 512 Kbytes.

El Tamaño del Bloque



Tamaño de bloque. Cuando se trae un bloque a la caché no solo se trae la palabra referenciada, sino también unas cuantas adyacentes. Así, debido al principio de localidad espacial, a medida que el tamaño del bloque aumenta, también aumenta la probabilidad de acierto. Sin embargo, esta probabilidad empezará a disminuir si el tamaño del bloque crece hasta el punto de que la probabilidad de acceder a otras palabras del bloque sea menor que la probabilidad de reutilizar la información que hay que reemplazar al traer un bloque grande a la caché.

Estas son las consecuencias negativas de los bloques de gran tamaño:

1. Al ser los bloques más grandes, se reduce el número de bloques de la caché. Ya que cada bloque que se trae a la caché reemplaza a otro, si tenemos pocos bloques en la caché la frecuencia de sustituciones de bloques será muy alta, es decir, que un bloque se reemplazará poco después de haberse cargado, lo cual, según el principio de localidad, no es nada bueno.
2. A medida que el tamaño del bloque se hace mayor, cada palabra adicional está más alejada de la que se ha referenciado, por lo que tiene menor probabilidad de ser referenciada en un futuro próximo.
3. Cuando se trae un bloque a la caché desde la memoria principal, cuanto mayor sea el bloque, más tiempo se tarda en leerlo.

...El Tamaño de Caché/Bloque

Tamaño bloque	Tamaño de la Caché				
	1K	4K	16K	64K	256K
16	15,05%	8,57%	3,94%	2,04%	1,09%
32	**13,34%**	7,24%	2,87%	1,35%	0,70%
64	13,76%	**7,00%**	**2,64%**	1,06%	0,51%
128	16,64%	7,78%	2,77%	**1,02%**	**0,49%**
256	22,01%	9,51%	3,29%	1,15%	0,49%

Tasa de faltas de caché, según el tamaño de bloque, para distintos tamaños de caché

Los datos que se muestran en estas tablas están extraídos del texto de Hennessy & Patterson, en el cual se indican los supuestos de los tiempos de acceso a memoria bajo los cuales se extraen estas conclusiones.

Mediante los datos comparativos que mostramos en esta tabla se comprueba que para un tamaño de caché dado, la tasa de faltas disminuye a medida que aumenta el tamaño del bloque. No obstante, a partir de cierto tamaño, esta tasa de faltas vuelve a crecer con el tamaño del bloque.

Al principio, al aumentar el tamaño de bloque disminuye la tasa de faltas debido al principio de la localidad espacial. Sin embargo, ya que a medida que aumenta el tamaño de bloque disminuye el número de estos en la caché, se empieza a hacer notar el perjuicio debido al principio de la localidad temporal, ya que resulta muy fácil que un bloque recién utilizado haya que expulsarlo para sustituirlo por otro.

Se aprecia que a medida que aumenta el tamaño de la caché se puede ir aumentando el tamaño de bloque. Esto quiere decir que, obviamente, el tamaño óptimo del bloque también depende del tamaño de la caché. También se aprecia que **para las cachés de gran tamaño, la mejora que se consigue aumentando el tamaño del bloque cada vez es más pequeña**, mientras que en las cachés de tamaño pequeño o medio, las mejoras son mucho más substanciales.

...El Tamaño de Caché/Bloque

Tamaño bloque	Tiempo por falta	Tamaño de la Caché 1K	4K	16K	64K	256K
16	42	7,321	4,599	2,655	1,859	1,458
32	44	**6,870**	**4,186**	**2,263**	1,594	1,308
64	48	7,605	4,360	2,267	**1,509**	**1,245**
128	56	10,318	5,357	2,551	1,571	1,274
256	72	16,847	7,847	3,369	1,828	1,353

Tiempo medio de acceso a memoria, según el tamaño de bloque, para distintos tamaños de caché

En esta tabla podemos ver los tiempos medios de acceso a memoria que se obtienen para los correspondientes casos de la tabla arriba mostrada.

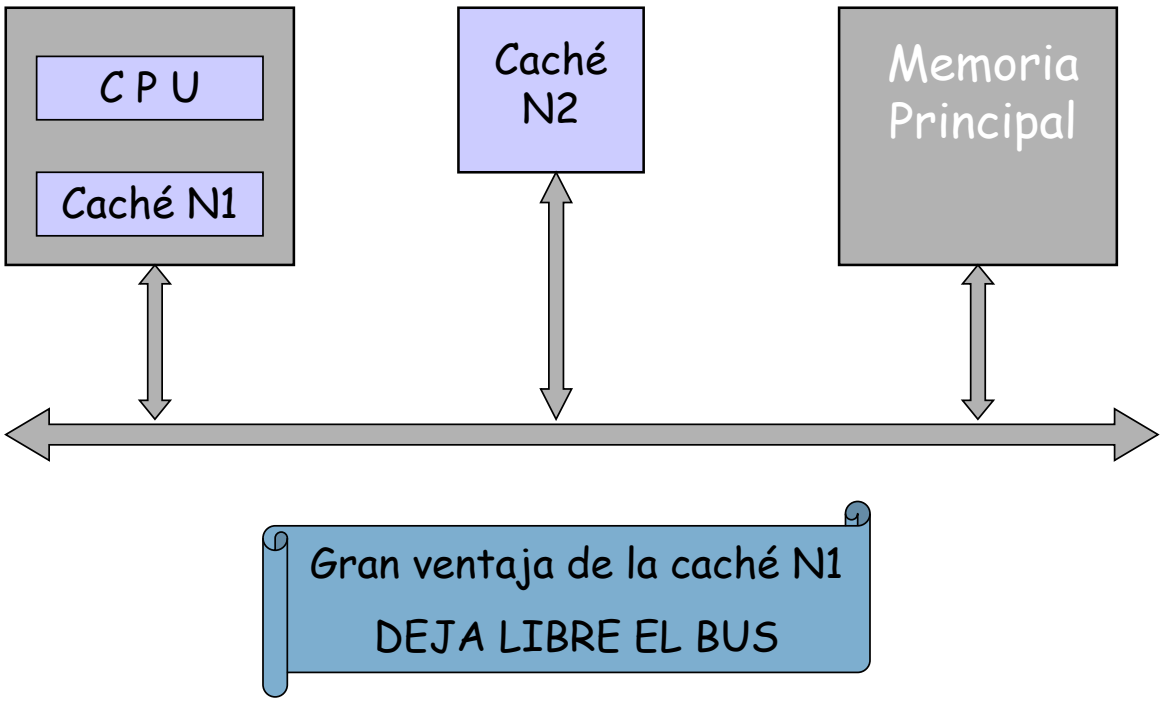
En la segunda columna se muestra el tiempo de acceso a memoria en caso de producirse una falta de caché. Como puede apreciarse, la penalización por falta de caché aumenta a medida que crece el tamaño y la complejidad de la caché.

También se puede apreciar que **el tiempo óptimo de acceso para cada tamaño de caché no se obtiene con el bloque más grande**, pues la tasa de aciertos disminuye cuando la probabilidad de que se vuelva a referenciar un bloque que se ha expulsado de la caché (localidad temporal) es mayor que la probabilidad de acceder a una palabra cercana a otra ya referenciada (localidad espacial).

Se advierte que a medida que aumenta el tamaño de la caché, disminuye el tiempo óptimo de acceso (en negrita), pero obsérvese también que **al principio, pequeños incrementos en el tamaño de la caché generan grandes mejoras en el tiempo**, mientras que al final, podemos ver que al pasar de una caché de 64K a otra de 256K, la mejora es ya muy pequeña, sobre todo comparada con el gran incremento de tamaño. Por eso actualmente los tamaños de las cachés oscilan entre 32 y 512 Kbytes.

Por último, nótese que **la combinación tamaño caché-bloque que ofrece el menor tiempo medio de acceso no es la que tiene la menor tasa de faltas**. Esto se debe a que en una caché grande, al incrementar el tamaño del bloque, mejora muy poco la tasa de aciertos y empeora mucho la penalización por falta de caché.

Número de Cachés



Cuando apareció la memoria caché cada sistema disponía de una única caché, pero últimamente resulta normal que se disponga de múltiples cachés. Hay dos cuestiones a considerar sobre el **número de cachés** de un sistema:

- ¿cuántos niveles de caché debe haber?
- ¿se debe compartir una única caché para instrucciones y datos o debe haber cachés separadas?

Niveles de caché.

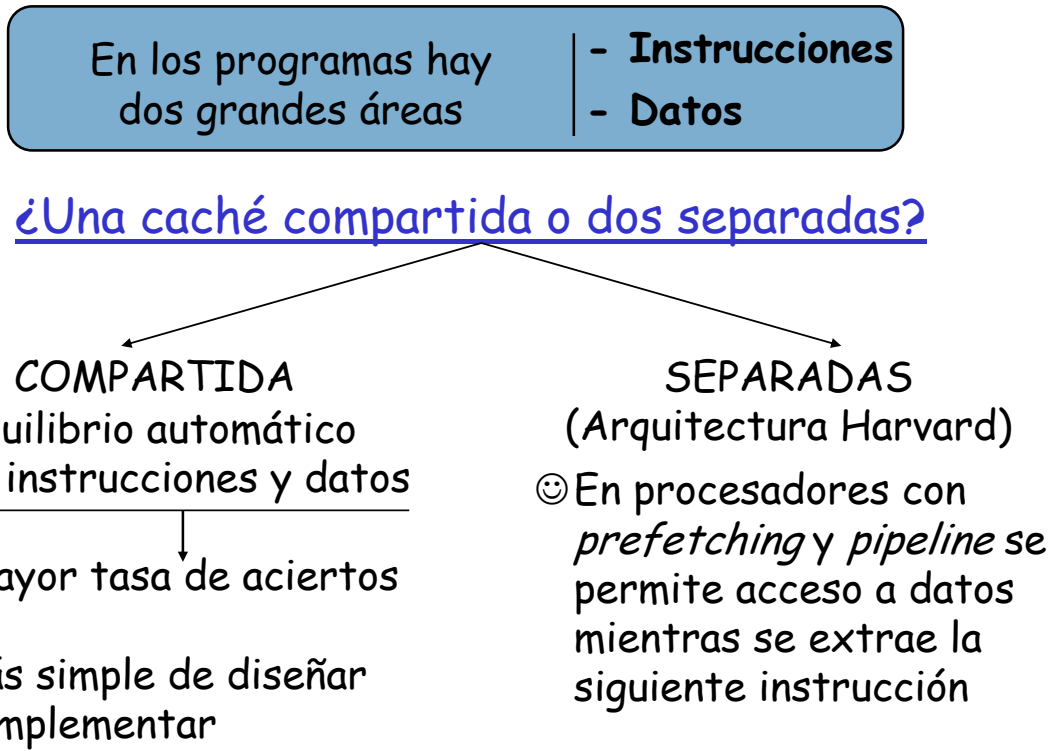
La tecnología actual de construcción de chips ha conseguido tal nivel de densidad de componentes que permite la inclusión de una memoria caché dentro del mismo chip de la CPU. Así, el acceso a una caché interna no solamente es más rápido que a una externa, sino que además reduce la actividad de la CPU con el bus, dejándolo libre para otros dispositivos que puedan necesitarlo, con lo que se incrementa la velocidad general del sistema.

Al incluir la caché dentro del mismo chip de la CPU surge la cuestión ¿merece la pena disponer de otra caché externa? La respuesta es sí. Por una parte ya hemos visto que no conviene que las cachés sean muy grandes, pues se pierde velocidad de acceso; por otra parte, resulta que el espacio disponible en el chip que contiene la CPU es muy reducido, lo cual limita mucho la capacidad de la caché interna. Así pues, las faltas de caché generan accesos a memoria principal con un tiempo de acceso que es varios órdenes de magnitud más lenta. Para paliar las faltas de la caché interna (primaria o de nivel 1) se instala una caché externa (secundaria o de nivel 2), más grande y más lenta, aunque mucho más rápida que la memoria principal.

Aunque la mejora en el tiempo de acceso que conlleva la caché de nivel 2 no es comparable a la del nivel 1, diversos estudios estadísticos demuestran que merece la pena la instalación de los dos niveles de memoria caché.

Los microprocesadores actuales, como Pentium e Itanium, tienen 2 o tres niveles de caché. La de nivel 1 es del orden de 32 Kbytes; la de nivel 2, de uno o dos MBytes; y la de nivel 3, de 6 a 24 MBytes.

...Número de Cachés



Cachés compartidas o separadas.

Los diseños de los primeros sistemas que incluyeron memoria caché disponían de una única caché para almacenar las referencias tanto a operandos como a instrucciones.

Hay dos ventajas potenciales para mantener la caché unificada:

1. Para un tamaño de caché dado, la versión unificada ofrece una mayor tasa de aciertos que la dividida, ya que se establece automáticamente el equilibrio necesario entre el espacio utilizado para las instrucciones y para los datos (operandos). Es decir, que si, por ejemplo, un modelo de ejecución tiende a realizar más referencias a instrucciones que a datos, la caché tenderá a llenarse con más instrucciones que con datos.
2. Es mucho más simple de diseñar e implementar una caché compartida.

A pesar de estas ventajas, los potentes procesadores de hoy día tienden a utilizar cachés separadas o divididas (arquitectura Harvard): una dedicada a instrucciones y otra a los datos u operandos, pues al disponer de la lectura adelantada de instrucciones (*prefetching*) y de ejecución segmentada de las instrucciones (*pipeline*), con cachés separadas se permite el acceso a un dato al mismo tiempo que la unidad de *prefetching* extrae la siguiente instrucción.