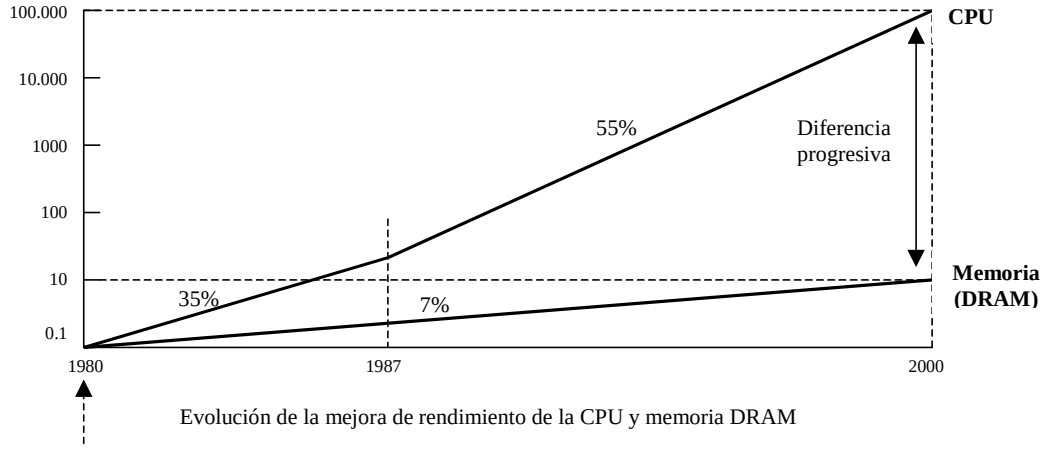
Tema 6: Memoria Caché.

Objetivos:

- Introducir la terminología y los principios básicos de funcionamiento de la memoria caché, resaltando la localidad referencial de los programas que explican su elevado rendimiento.
- Analizar las alternativas de diseño que determinan el comportamiento de una caché, resaltando la función de correspondencia, las políticas de búsqueda y sustitución de bloques, y el mantenimiento de la coherencia con memoria principal en las escrituras.
- Estudiar los factores que más influencia tienen en el rendimiento de una caché junto a las alternativas de diseño hardware y software que permiten optimizar dichos factores.
- Analizar el sistema de memoria caché de algunos procesadores, especialmente el del ARM.

Contenido:

La velocidad de la memoria se ha distanciado progresivamente de la velocidad de los procesadores. En la figura siguiente se muestran las gráficas de la evolución experimentada por el rendimiento de las CPUs y las memoria DRAM (soporte de la memoria principal de los computadores actuales) en los últimos años. Las curvas muestran que el rendimiento de la CPU aumentó un 35% anual desde 1980 hasta 1987; y un 55% anual a partir de ese año. En cambio la memoria ha mantenido un crecimiento sostenido del 7% anual desde 1980 hasta la fecha. Esto significa que si se mantiene la tendencia, el diferencial de rendimiento no sólo se mantendrá sino que aumentará en el futuro. Para equilibrar esta diferencia se viene utilizando una solución arquitectónica: *la memoria caché*



Evolución de la mejora de rendimiento de la CPU y memoria DRAM

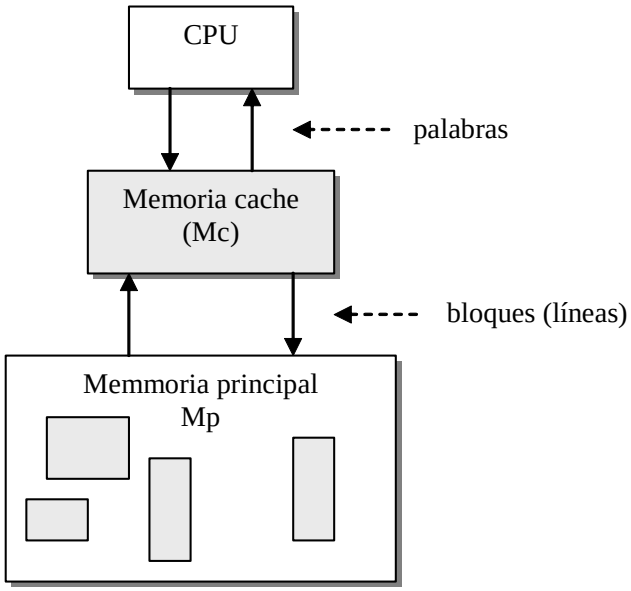
La memoria caché es una memoria pequeña y rápida que se interpone entre la CPU y la memoria principal para que el conjunto opere a mayor velocidad. Para ello es necesario mantener en la caché aquellas zonas de la memoria principal con mayor probabilidad de ser referenciadas. Esto es posible gracias a la propiedad de localidad de referencia de los programas.

1.1. Localidad de referencia: temporal y espacial

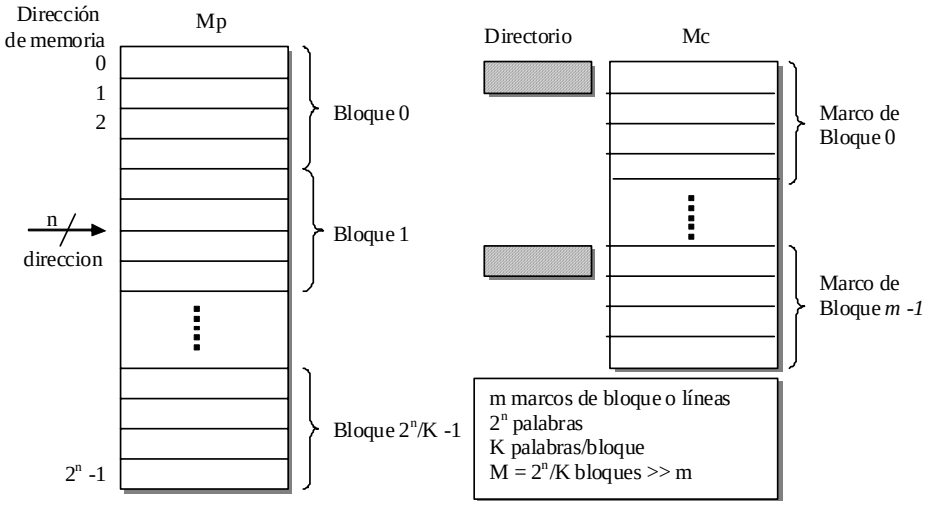
Los programas manifiestan una propiedad que se explota en el diseño del sistema de gestión de memoria de los computadores en general y de la memoria caché en particular, la *localidad de referencias*: los programas tienden a reutilizar los datos e instrucciones que utilizaron recientemente. Una regla empírica que se suele cumplir en la mayoría de los programas revela que gastan el 90% de su tiempo de ejecución sobre sólo el 10% de su código. Una consecuencia de la localidad de referencia es que se puede predecir con razonable precisión las instrucciones y datos que el programa utilizará en el futuro cercano a partir del conocimiento de los accesos a memoria realizados en el pasado reciente. La localidad de referencia se manifiesta en una doble dimensión: temporal y espacial.

Localidad temporal: las palabras de memoria accedidas recientemente tienen una alta probabilidad de volver a ser accedidas en el futuro cercano. La localidad temporal de los programas viene motivada principalmente por la existencia de bucles.

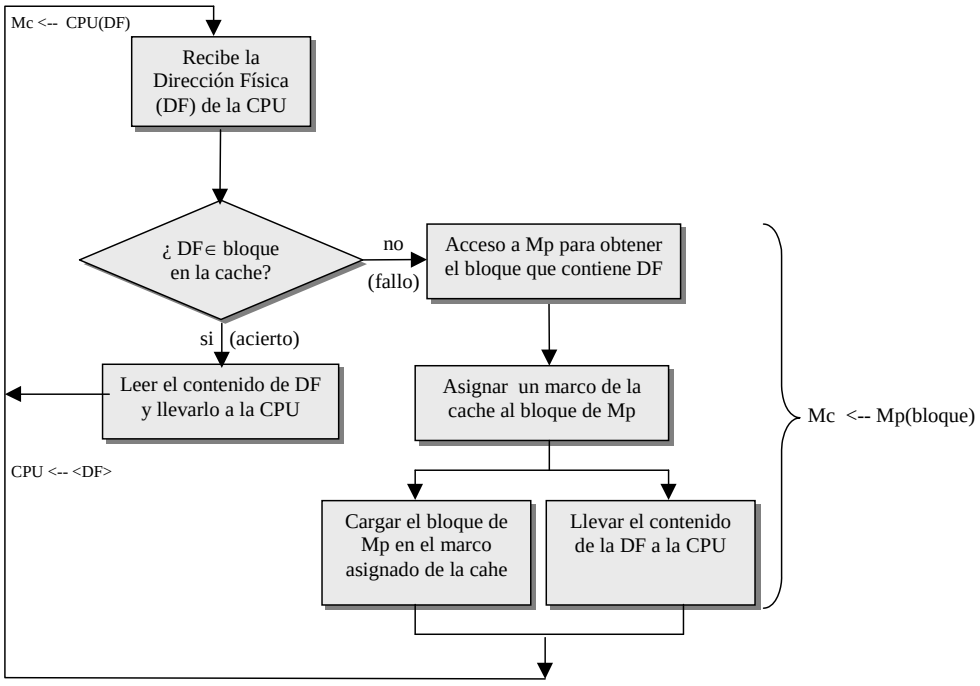
Localidad espacial: las palabras próximas en el espacio de memoria a las recientemente referenciadas tienen una alta probabilidad de ser también referenciadas en el futuro cercano. Es decir, que las palabras próximas en memoria tienden a ser referenciadas juntas en el tiempo. La localidad espacial viene motivada fundamentalmente por la linealidad de los programas (secuenciamiento lineal de las instrucciones) y el acceso a las estructuras de datos regulares.



Para implementar el mecanismo de actualización de la caché con los datos con mayor probabilidad de ser refernciados se divide la memoria principal en bloques de un número de bytes (4,8,16 etc.) y la caché en marcos de bloque o líneas de igual tamaño. El bloque será, pues, la unidad de intercambio de información entre la memoria principal y la caché, mientras que entre la caché y la CPU sigue siendo la palabra. El *directorio* contiene la información de qué bloques de *Mp* se encuentran ubicados en *Mc*



El funcionamiento de la memoria caché se puede resumir en el diagrama de flujo de la siguiente figura. En él se describe el proceso de traducción de la dirección física procedente de la CPU (en el supuesto que el procesador no disponga de memoria virtual o esté desactivado) en el dato ubicado en la posición de memoria determinada por dicha dirección:



1.2. Tasa de aciertos y tasa de fallos:

Cuando una dirección se presenta en el sistema caché pueden ocurrir dos cosas:

- Acierto de caché (*hit*): el contenido de la dirección se encuentre en un bloque ubicado en una línea de la caché.
- Fallo de caché (*miss*): el contenido de la dirección no se encuentre en ningún bloque ubicado en alguna línea de la caché.

Si en la ejecución de un programa se realizan *Nr* referencias a memoria, de las que *Na* son aciertos caché y *Nf* fallos caché, se definen los siguientes valores:

- Tasa de aciertos: $Ta = Na / Nr$
- Tasa de fallos: $Tf = Nf / Nr$

Evidentemente se cumple: $Ta = 1 - Tf$

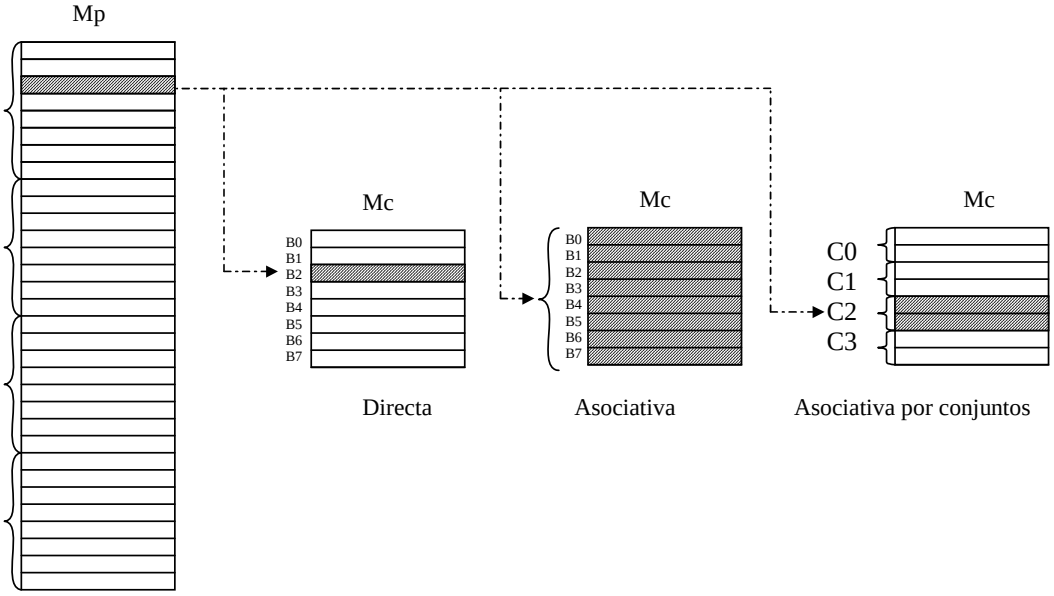
2. Elementos de diseño.

A la hora de diseñar un sistema de memoria caché hay que elegir entre una serie de alternativas para cada uno de los siguientes elementos de diseño:

- Función de correspondencia: determina las posibles líneas de la caché (marcos de bloque) en las que se puede ubicar un determinado bloque de la memoria principal que ha sido referenciado por el programa y hay que llevarlo a memoria caché.
- Algoritmo de sustitución: determina el bloque que hay que desubicar de una línea de la caché cuando ésta está llena y hay que ubicar un nuevo bloque.
- Política de escritura: determina la forma de mantener la coherencia entre memoria caché y memoria principal cuando se realizan modificaciones (escrituras)
- Política de búsqueda de bloques: determina la causa que desencadena la llevada de un bloque a la caché (normalmente un fallo en la referencia)
- Cachés independientes para datos e instrucciones: frente a cachés unificadas.

2.1. Función de correspondencia

Existen tres funciones de correspondencia para definir la posible ubicación de un bloque de memoria principal (Mp) en la memoria caché (Mc): *directa*, *asociativa* y *asociativa por conjuntos*. En el primer caso un bloque de Mp sólo puede ubicarse en una línea de la caché, aquella que coincide con el bloque cuando superponemos Mc sobre Mp respetando fronteras de Mc, es decir, sobre espacios de Mp que son múltiplos del tamaño de Mc. En la correspondencia asociativa un bloque puede ubicarse en cualquier línea de Mc. Finalmente, la correspondencia asociativa por conjuntos es un compromiso entre las dos anteriores.

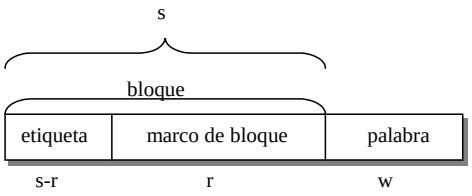


2.1.1. Correspondencia directa

En la *correspondencia directa* el bloque *Bj* de *Mp* se puede ubicar sólo en el marco de bloque o línea *MBi* que cumple la siguiente relación $i = j \bmod m$, donde m es el número total de líneas que tiene la caché. En la tabla siguiente se especifica el conjunto de bloques que se pueden ubicar en una línea de *Mc*:

Bloques de Mp	Marcos de bloque de Mc
0, m, 2m, ... , 2^{s-1}m	0
1, m+1, 2m+1,..., 2^{s-1}m+1	1
.....	...
m-1, 2m-1, 3m-1,..., 2^{s}m-1	m-1

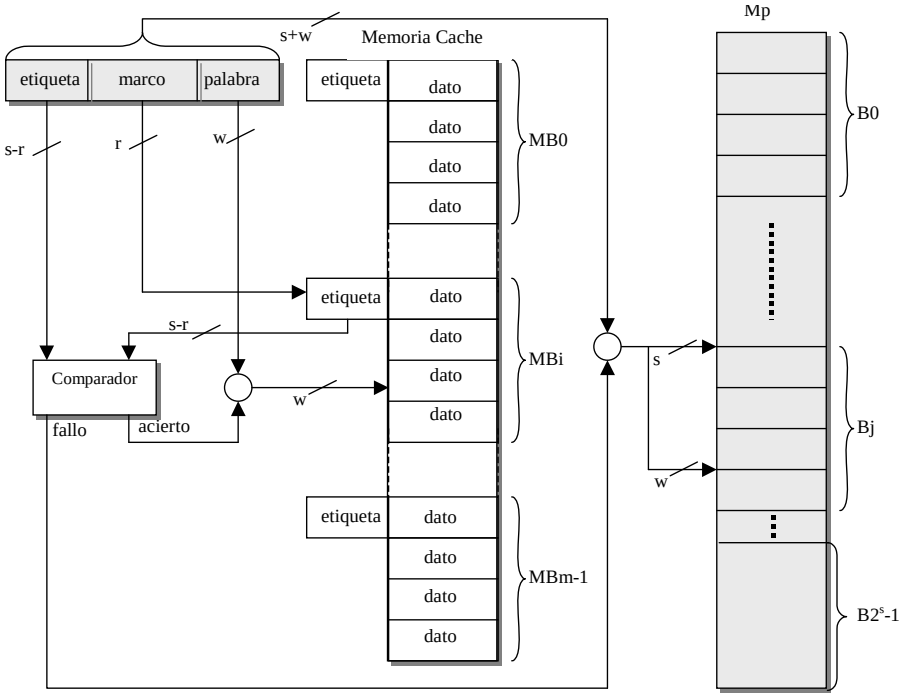
Interpretación de una dirección física en correspondencia directa:



2^w palabras/bloque
2^s bloques de Mp
2^r marcos de bloque en Mc (2^r = m)
2^{s-r} veces contiene Mp a Mc

Los *s - r* bits de la *etiqueta* diferenciarán a cada uno de los bloques de Mp que pueden ubicarse en el mismo marco de bloque de Mc. El directorio caché en correspondencia directa contendrá un registro de *s - r* bits por cada marco de bloque para contener la etiqueta del bloque ubicado en ese momento en dicho marco.

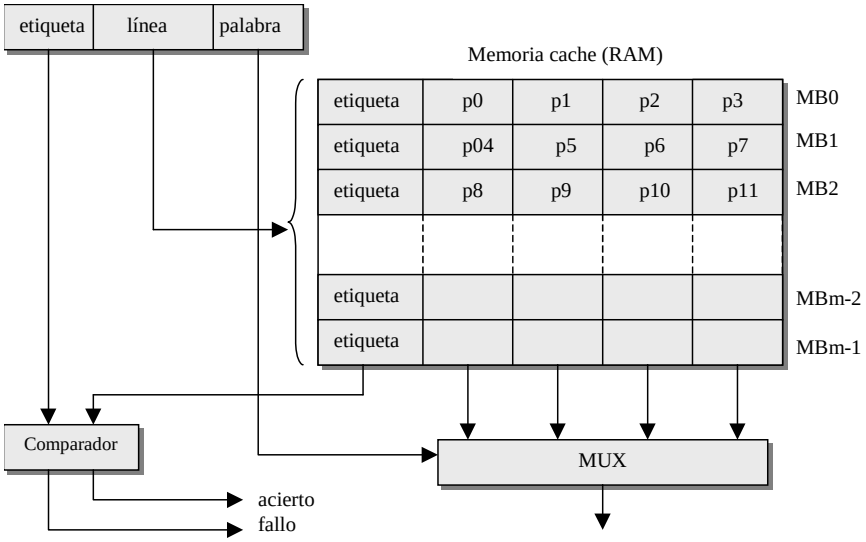
El mecanismo de obtención del contenido de una dirección física en cachés con correspondencia directa podemos resumirlo en el siguiente esquema:



Ejemplo: para los tres tipos de correspondencia utilizaremos los siguientes datos:

- Tamaño de bloque K = 4 bytes = 2^2 => w = 2
- Tamaño de Mc = 64 KBytes = 2^{16} Bytes = 2^{14} marcos de bloque => r = 14
- Tamaño de Mp = 16 MBytes = 2^{24} Bytes = 2^{22} bloques => s = 22

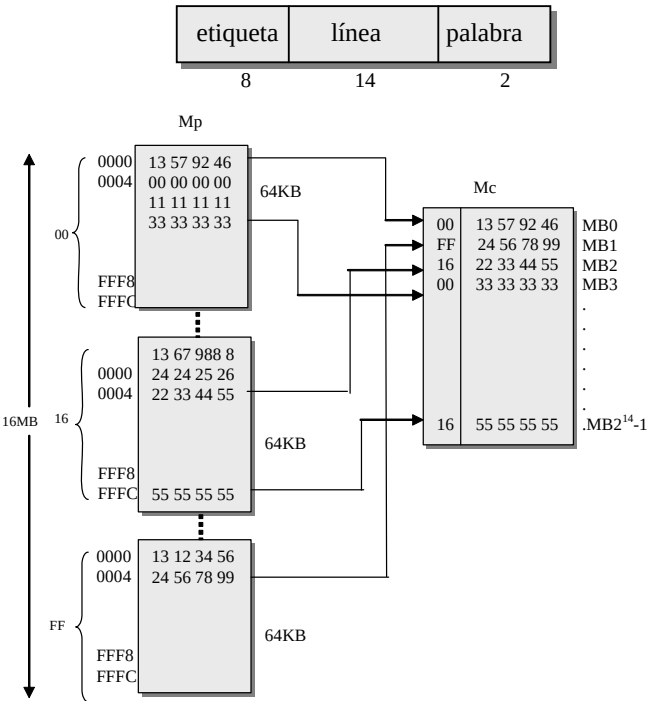
Una Mc de correspondencia directa se puede implementar sobre una RAM con longitud de palabra suficiente para ubicar un bloque y los bits de etiqueta (directorio), tal como se muestra en la siguiente figura:



En una operación de lectura se lee la palabra completa de la RAM, es decir, la línea y la etiqueta. Si la etiqueta leída coincide con la procedente de la dirección física, significa que la línea contiene la palabra de *Mp* referenciada por dicha dirección física: se produce un *acierto* de caché. En este caso con los *w* bits de *palabra* se selecciona la palabra referenciada dentro de la línea.

Si no coinciden las etiquetas, significa que *Mc* no contiene el bloque de *Mp* al que pertenece la palabra referenciada, por lo que se produce un *fallo* de caché.

Parte de un posible contenido de *Mc* en un instante determinado para el ejemplo anterior podría ser el siguiente:

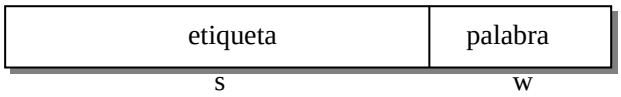


Se ha dibujado Mp dividididda en zonas consecutivas de tamaño 64 KB (igual que el tamaño de Mc) para facilitar la correspondencia de los bloques de Mp y los marcos de bloque de Mc.

2.1.2. Correspondencia asociativa

En la *correspondencia asociativa* un bloque *Bj* de Mp se puede ubicar en cualquier marco de bloque de *Mc*.

Interpretación de una dirección física en correspondencia asociativa:

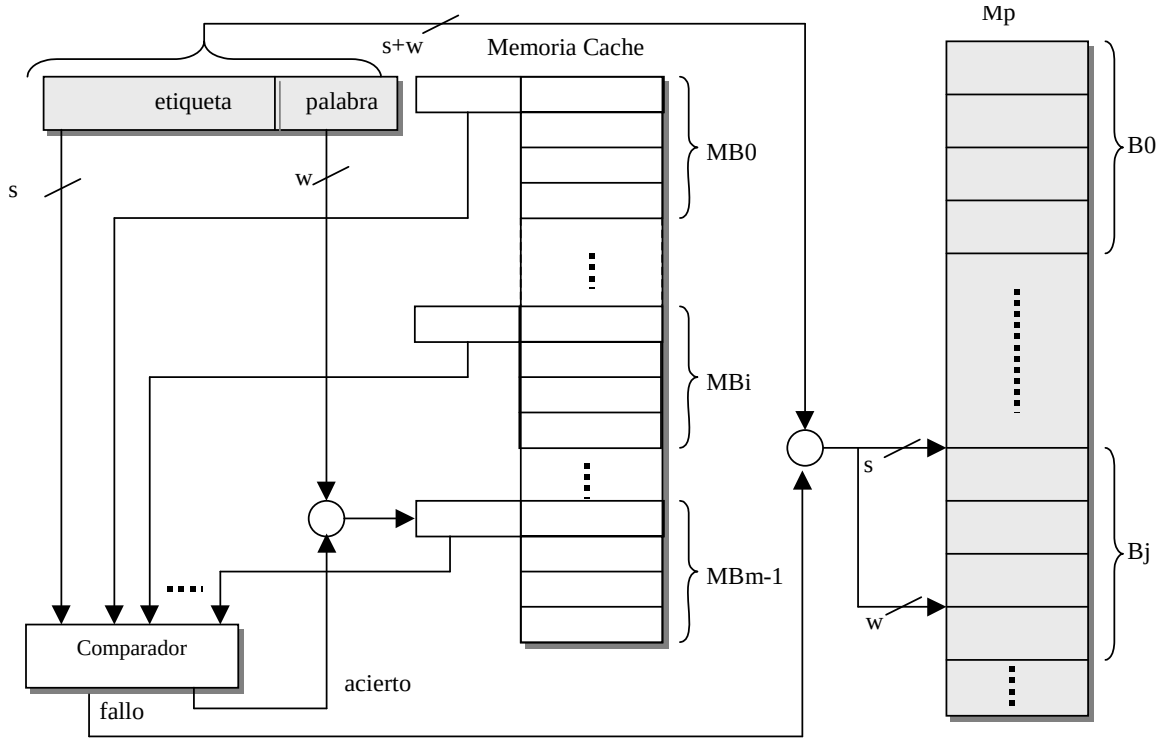


2^s bloques de Mp

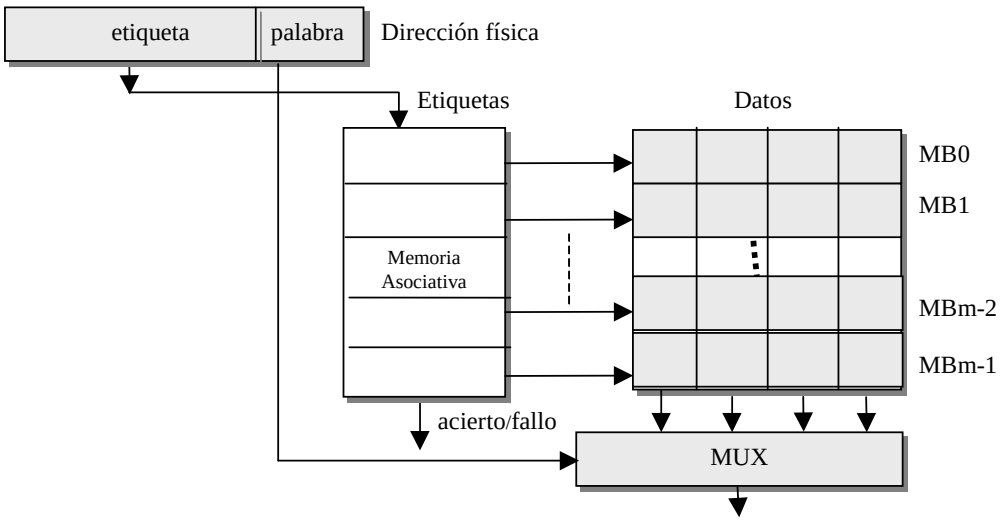
2^r marcos de bloque de Mc

En este caso la etiqueta tendrá *s* bits para poder diferenciar a cada uno de los bloques de *Mp* (todos) que pueden ubicarse en el mismo marco de bloque de Mc. El directorio caché en correspondencia asociativa contendrá, pues, un registro de *s* bits por cada marco de bloque para contener la etiqueta del bloque ubicado en ese momento en dicho marco.

El mecanismo de obtención del contenido de una dirección física en cachés con correspondencia asociativa podemos resumirlo en el siguiente esquema:



El directorio de una *Mc* de correspondencia asociativa se puede implementar con una memoria asociativa con tantas palabras como líneas tenga *Mc*. Las líneas se soportan sobre un *array* de memoria con longitud de palabra suficiente para ubicar un bloque, tal como se muestra en la siguiente figura:

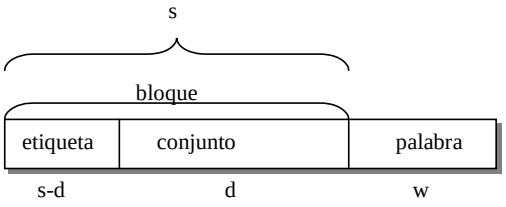


La memoria asociativa contiene las direcciones de todos los bloques de *Mp* ubicados en cada momento en *Mc*. Opera realizando una comparación simultánea de su contenido con el campo de etiqueta de la dirección física.

2.1.3. Correspondencia asociativa por conjuntos

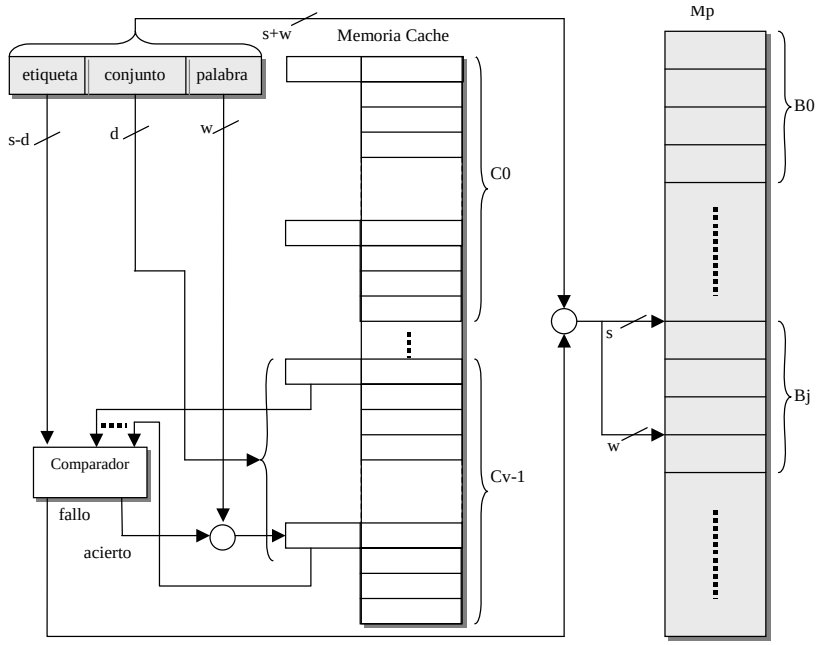
En la correspondencia asociativa por conjuntos las líneas de *Mc* se agrupan en $v=2^d$ conjuntos con *k* líneas/conjunto o *vías* cada uno. Se cumple que el número total de marcos de bloque (líneas) que tiene la caché m = v*k. Un bloque *Bj* de *Mp* se puede ubicar sólo en el conjunto *Ci* de *Mc* que cumple la siguiente relación $i = j \bmod v$.

Interpretación de una dirección física en correspondencia asociativa por conjuntos:

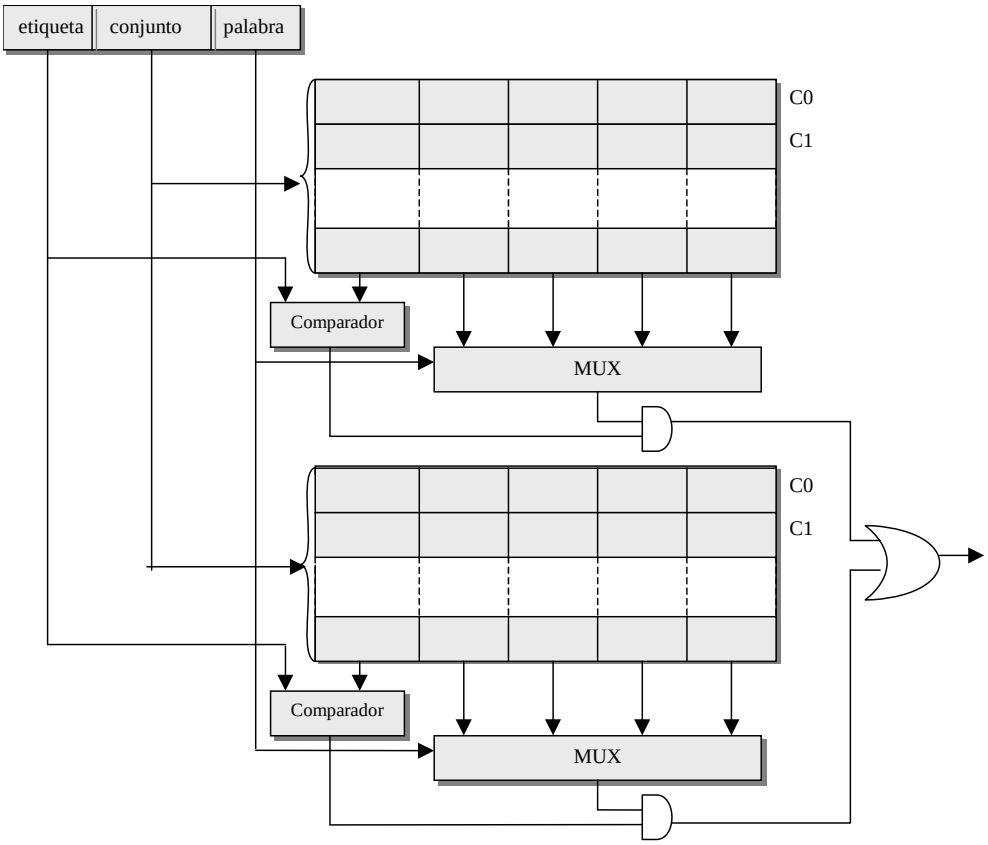


En este caso la etiqueta tendrá $s - d$ bits para poder diferenciar a cada uno de los bloques de *Mp* que pueden ubicarse en el mismo conjunto de *Mc*. El directorio caché en correspondencia asociativa por conjuntos contendrá, pues, un registro de $s - d$ bits por cada línea de *Mc*.

El esquema lógico de acceso a una caché de correspondencia asociativa por conjuntos se muestra en la siguiente figura:



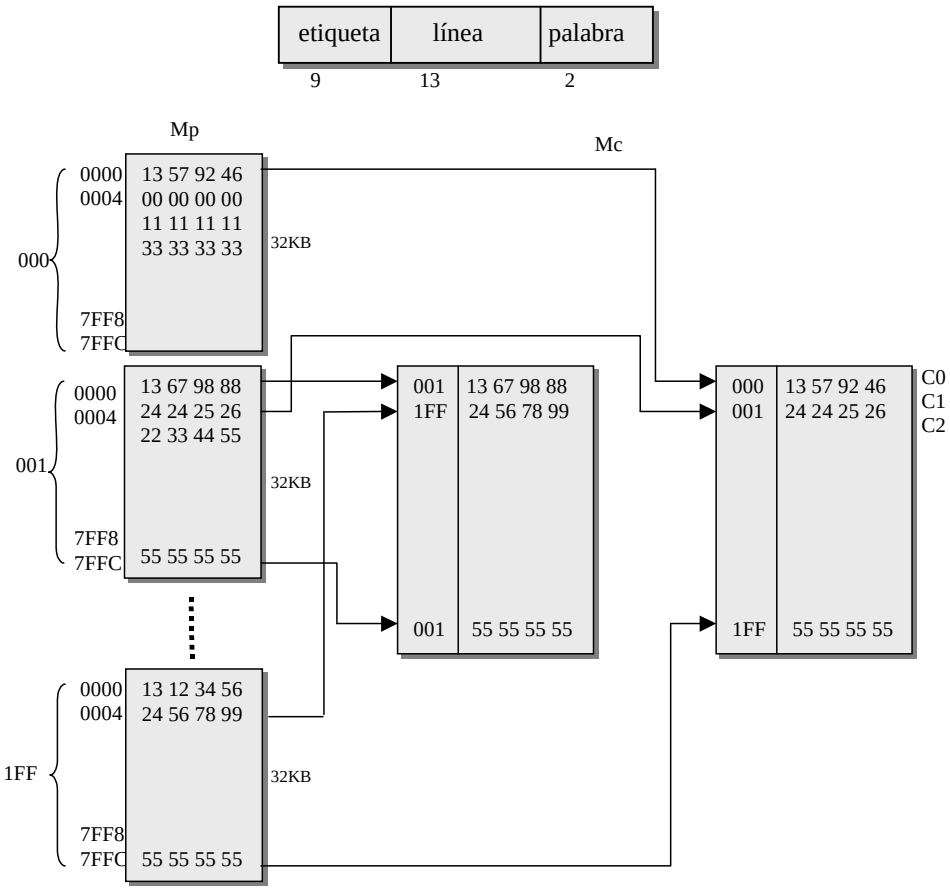
Una *Mc* de correspondencia asociativa por conjuntos de *v* conjuntos se puede implementar como *k* módulos de correspondencia directa en paralelo cada uno con *v* líneas. Los conjuntos lo formarían las líneas que ocupan idéntica posición en cada módulo:



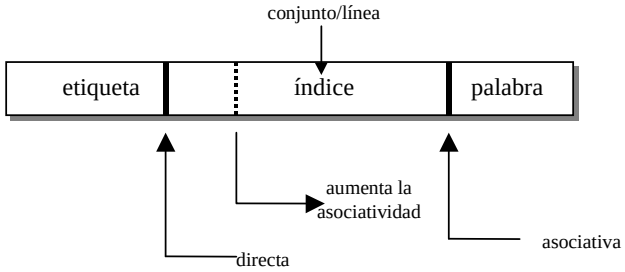
Si dibujamos *Mp* dividididda en zonas consecutivas de tamaño 32 KB (con tantos bloques como conjuntos tiene *Mc*) para facilitar la correspondencia de los bloques de *Mp* y los marcos de

bloque de Mc, podremos representar fácilmente parte de un posible contenido de *Mc* con correspondencia asociativa por conjuntos de dos vías en un instante determinado para el ejemplo anterior.

$k = 2 \Rightarrow v = m/k = 2^r/2 = 2^{14}/2 = 2^{13} \Rightarrow d = 13$
$w = 2$
$s = 22 \Rightarrow s - d = 9$



Las tres funciones de correspondencia se pueden ver en realidad como una sola, la asociativa por conjunto, siendo las otras dos correspondencias casos extremos de ella:



Si $k = 1 \Rightarrow m = v \Rightarrow$ asociativa por conjuntos de 1 vía = directa

Si $v = 1 \Rightarrow m = k \Rightarrow$ asociativa por conjuntos de m vías = asociativa

2.2. Algoritmos de sustitución

Como hemos visto, el espacio de ubicación de un bloque en *Mc* depende de la función de correspondencia. La correspondencia directa reduce el espacio a un marco de bloque, por lo que no queda alternativa de sustitución cuando hay que ubicar un nuevo bloque. En las correspondencias asociativa y asociativa por conjuntos hay que decidir qué bloque sustituir. En la primera la elección se tiene que realizar sobre cualquier bloque presente en *Mc*, mientras que en la segunda se reduce al conjunto único donde puede ubicarse el nuevo bloque. Las políticas de reemplazamiento más utilizadas son cuatro.

2.2.1. Aleatoria

- Se escoge un bloque al azar

2.2.2. LRU (*Least Recently Used*)

- Se sustituye el bloque que hace más tiempo que no ha sido referenciado
- Algoritmo: se asocian contadores a los marcos de bloque y

  Acierto: Si se referencia *MBi*

  $\forall MBk : contador(MBk) \leq contador(MBi) \implies contador(MBk) := contado(MBk) + 1$

  $contador(MBi) = 0$

  Fallo: Cuando se produce un fallo se sustituye el $MBi : contador(MBi) = MAXIMO$
- Para una memoria asociativa de dos vías basta con añadir un bit de uso a cada marco de bloque. Cuando un marco de bloque es referenciado su bit de uso se pone a 1 y el del marco del mismo conjunto a 0. Cuando hay que sustituir se elige el marco de bloque con bit de uso igual a 0.

2.2.3. FIFO (*First In First Out*)

- Se sustituye aquel bloque que ha estado más tiempo en la caché (independientemente de sus referencias)
- Se puede implementar con una cola

2.2.4. LFU (*Least Frequently Used*)

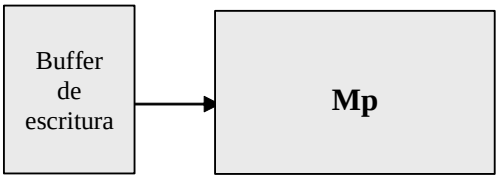
- Se sustituye aquel bloque que ha experimentado menos referencias
- Se puede implementar asociando un contador a cada marco de bloque

2.3. Política de escritura

Determina la forma de actualizar Mp cuando se realizan operaciones de escritura. Hay que diferenciar dos casos: cuando la posición de memoria sobre la que se va a escribir está en Mc (acierto) y cuando no lo está (fallo). Frente a aciertos en la caché existen dos alternativas: escritura directa y postescritura. Y frente a fallos en la caché otras dos: asignación en escritura y no asignación.

2.3.1. Escritura directa o inmediata (*write through*)

- Todas las operaciones de escritura se realizan en Mc y Mp
- Inconveniente: genera un tráfico importante a Mp
- Solución: utilización de un buffer de escritura (alpha 21064)



2.3.2. Postescritura (*copy back*)

- Las actualizaciones se hacen sólo en *Mc*

- Se utiliza un bit de actualización asociado a cada marco de bloque para indicar la escritura del marco en Mp cuando es sustituido por la política de reemplazamiento
- Inconveniente: inconsistencia temporal entre Mc y Mp ==> complicación del acceso de la E/S a memoria que debe realizarse a través de Mc.

2.3.3. Asignación en escritura (*write allocate*)

- El bloque se ubica en Mc cuando ocurre el fallo de escritura y a continuación se opera como en un acierto de escritura, es decir, con *wirte through* o *copy back*

2.3.4. No asignación en escritura (*No write allocate*)

- El bloque se modifica en Mp sin cargarse en Mc

2.4. Política de búsqueda

Determina las condiciones que tienen que darse para buscar un bloque de Mp y llevarlo a una línea de Mc. Existen dos alternativas principales: por demanda y anticipativa.

2.4.1. Por demanda

- Se lleva un bloque a Mc cuando se referencia desde la CPU alguna palabra del bloque y éste no se encuentra en Mc

2.4.2. Anticipativa (prebúsqueda)

- Prebúsqueda siempre: la primera vez que se referencia el bloque Bi se busca también Bi+1
- Prebúsqueda por fallo: cuando se produce un fallo al acceder al bloque Bi se buscan los bloques Bi y Bi+1

2.5. Clasificación de los fallos caché

Los fallos de la caché se pueden clasificar en tres tipos:

- Forzosos: producidos por el primer acceso a un bloque
- Capacidad: producidos cuando Mc no puede contener todos los bloques del programa
- Conflicto: (en correspondencia directa o asociativa por conjuntos) producidos por la necesidad de ubicar un bloque en un conjunto lleno cuando Mc no está completa.

3. Factores que determinan el rendimiento de la memoria caché.

Si frente a un fallo, el bloque de Mp se lleva a Mc al tiempo que la palabra referenciada del bloque se lleva (en paralelo) a la CPU, el tiempo de acceso a memoria durante la ejecución de un programa será :

$Tacceso = Na * Tc + Nf * Tp$ donde:

Na es el número de referencias con acierto

Nf es el número de referencias con fallo

Tc es el tiempo de acceso a una palabra de *Mc*

Tp es el tiempo de acceso a un bloque de *Mp*

El tiempo de acceso a un bloque de Mp, Tp, constituye la componente principal del tiempo total de penalización por fallo.

El tiempo de acceso medio durante la ejecución del programa valdrá:

$Tacceso\_medio = Tacceso / Nr = Ta*Tc + Tf*Tp$ donde:

$Ta = Na / Nr$ es la tasa de aciertos

$Tf = Na / Nr$ es la tasa de fallos

Nr es el número total de referencias a memoria

A la primera componente se le denomina $Tacierto = Ta*Tc$,

En cambio, si frente a un fallo, el bloque de Mp se lleva primero a Mc y después se lee la palabra referenciada de Mc y se lleva a la CPU, el tiempo de acceso a memoria durante la ejecución de un programa será :

$Tacceso = Nr * Tc + Nf * Tp$ y

$Tacceso\_medio = Tacceso / Nr = Tc + Tf*Tp$

En este caso $Tacierto = Tc$

De cualquiera de las expresiones anteriores podemos deducir que para mejorar el rendimiento de una caché, podemos actuar sobre tres términos, dando lugar a tres tipos de optimizaciones:

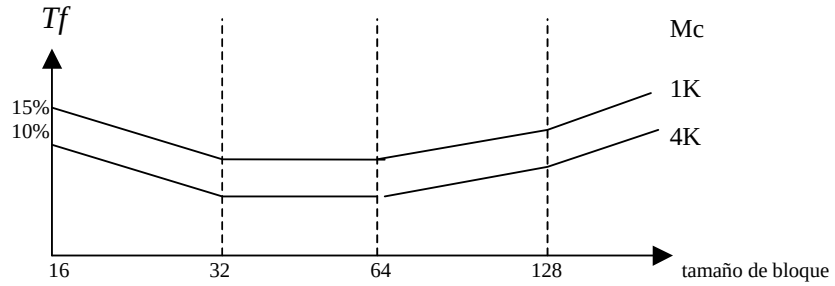
1. Reducción de la tasa de fallos, *Tf*
2. Reducción de la penalización de los fallos, *Tp*
3. Reducción del tiempo de acierto, *Tacierto*

3.1. Reducción de la tasa de fallos

La tasa de fallos podemos reducirla con las siguientes alternativas:

3.1.1. Aumento del tamaño del bloque

- Al aumentar el tamaño de bloque disminuye la tasa de fallos iniciales (forzosos) porque mejora la localidad espacial.
- Sin embargo con el aumento del tamaño de bloque aumenta la penalización de fallos, ya que el tiempo de lectura y transmisión serán mayores si los bloques son mayores.

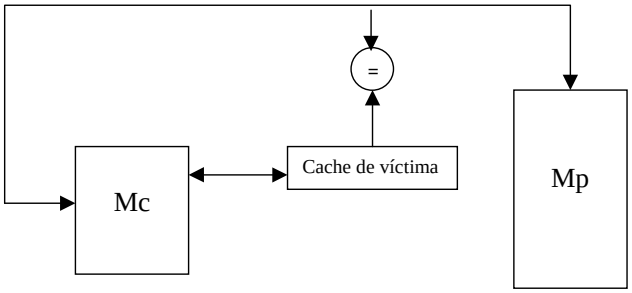


3.1.2. Aumento de la asociatividad

- Experimentalmente se comprueba que una caché asociativa por conjuntos de 8 vías es tan eficiente (tasa de fallos) como una caché completamente asociativa.
- Una caché de correspondencia directa de tamaño N tiene aproximadamente la misma tasa de fallos que una asociativa por conjuntos de 2 vías de tamaño N/2
- Al aumentar la asociatividad se incrementa el ciclo de reloj y por tanto *Tacierto*

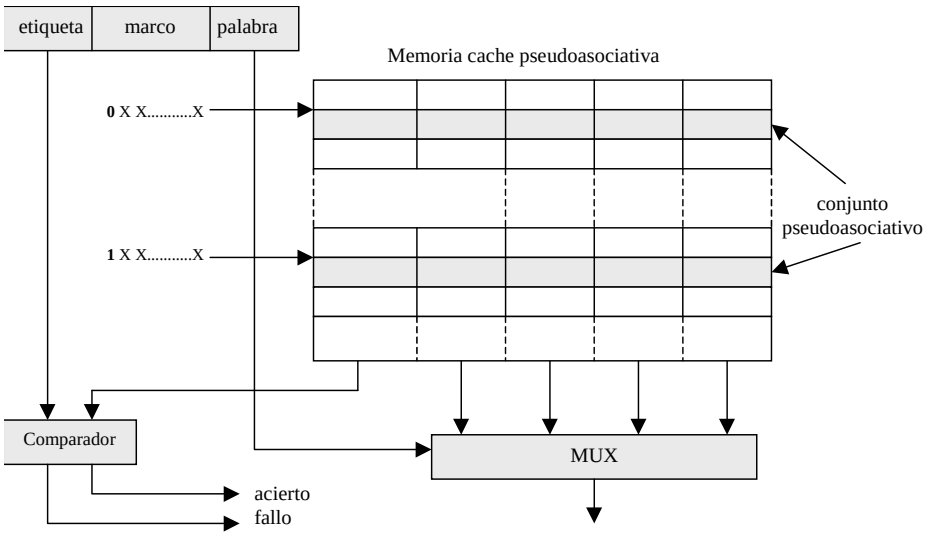
3.1.3. Utilización de una caché de víctimas

- Se añade una pequeña caché completamente asociativa entre Mc y su camino hacia Mp para contener sólo los bloques descartados (sustituidos) por un fallo (víctimas)
- Ante un fallo se comprueba si el bloque está en la caché de víctima antes de acudir a Mp
- Reduce los fallos de conflicto fundamentalmente en cachés pequeñas con correspondencia directa.



3.1.4. Cachés pseudoasociativas

Se trata de cachés de correspondencia directa que con una ligera modificación se pueden comportar como asociativas. Para ello se permite que un bloque de Mp se pueda ubicar en dos (pseudoasociativa de 2 vías) marcos de bloque de Mc, el que le corresponde (por la correspondencia directa) y el que resulta de conmutar el bit más significativo de la dirección del bloque, tal como se indica en el siguiente esquema:



Analicemos el rendimiento de una Mc pseudoasociativa:

$Tiempo\_acceso\_medio_{pseudo} = Tiempo\_acierto_{pseudo} + Tasa\_fallos_{pseudo} * Penalización\_fallos_{pseudo}$

$Tasa\_fallos_{pseudo} = Tasa\_fallos_{2vías}$

$Penalización\_fallos_{pseudo} = Penalización\_fallos_{directa}$

$Tiempo\_acierto_{pseudo} = Tiempo\_acierto_{directa} + Tasa\_aciertos\_alternativos_{pseudo} * 2$

$Tasa\_aciertos\_alternativos_{pseudo} = Tasa\_aciertos_{2vías} - Tasa\_aciertos_{directa} =$

$(1 - Tasa\_fallos_{2vías}) - (1 - Tasa\_fallos_{directa}) = Tasa\_fallos_{directa} - Tasa\_fallos_{2vias}$

$Tiempo\_acceso\_medio_{pseudo} = Tiempo\_acierto_{directo} + (Tasa\_fallos_{directa} - Tasa\_fallos_{2vias}) * 2 +$

$Tasa\_fallos_{2vías} * Penalización\_fallos_{directa}$

$Tasa\_aciertos\_alternativos_{pseudo}$ corresponde a la tasa de los aciertos que no encuentran el bloque en la primera línea alternativa de las dos en las que puede albergarse con la modificación (*pseudo*) introducida. Va multiplicado por 2 porque en este caso se requieren dos accesos a la caché, en el primero accede a la primera línea y en el segundo a la línea *pseudo* alternativa.

Ejemplo:

Tamaño (caché)	grado asociatividad	tasa de fallos	penalización fallo	Tiempo de acierto
2 K	1	0,098	50	1
2 K	2	0,076	-	-

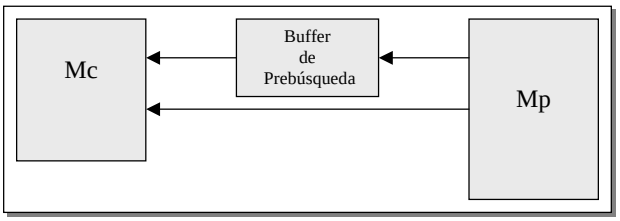
$Tiempo\_acceso\_medio_{pseudo} = 1 + (0{,}098 - 0{,}076)*2 + 0{,}076*50 = 4{,}844$

$Tiempo\_acceso\_medio_{directa}\ 1 + 0{,}098*50 = 5{,}90 > 4{,}844$

3.1.5. Prebúsqueda de instrucciones y datos

- La prebúsqueda de instrucciones y/o datos antes de ser demandados por la caché disminuye la tasa de fallos.

- Las instrucciones o datos prebuscados son llevados directamente a la caché o a un buffer externo que es accedido a mayor velocidad que Mp



- El Alpha AXP 21064 pre-busca dos bloques cuando ocurre un fallo, el que contiene la palabra causante del fallo y el siguiente. El primero lo coloca en MC y el segundo en el *buffer de prebúsqueda*
- Experimentalmente se ha comprobado que un buffer de prebúsqueda simple elimina el 25 % de los fallos de una caché de datos con correspondencia directa de 4 KB

3.1.6. Prebúsqueda controlada por el compilador

- Utiliza instrucciones explícitas de prebúsqueda del tipo *prefetch(dato)* que el compilador utiliza para optimizar los programas después de realizar un análisis de sus sentencias.
- La prebúsqueda se realiza al tiempo que el procesador continúa la ejecución del programa, es decir, la prebúsqueda se hace en paralelo con la ejecución de las instrucciones.
- Los bucles son las construcciones más apropiadas para que el compilador genere prebúsqueda

Ejemplo

Supongamos que en una caché de 8 KB de correspondencia directa y bloques de 16 bytes se ejecuta el siguiente programa:

```
for (i = 0; i < 3; i = i + 1)
    for (j = 0; j < 100; j = j + 1)
            a[i][j] = b[j][0] * b[j+1][0];
```

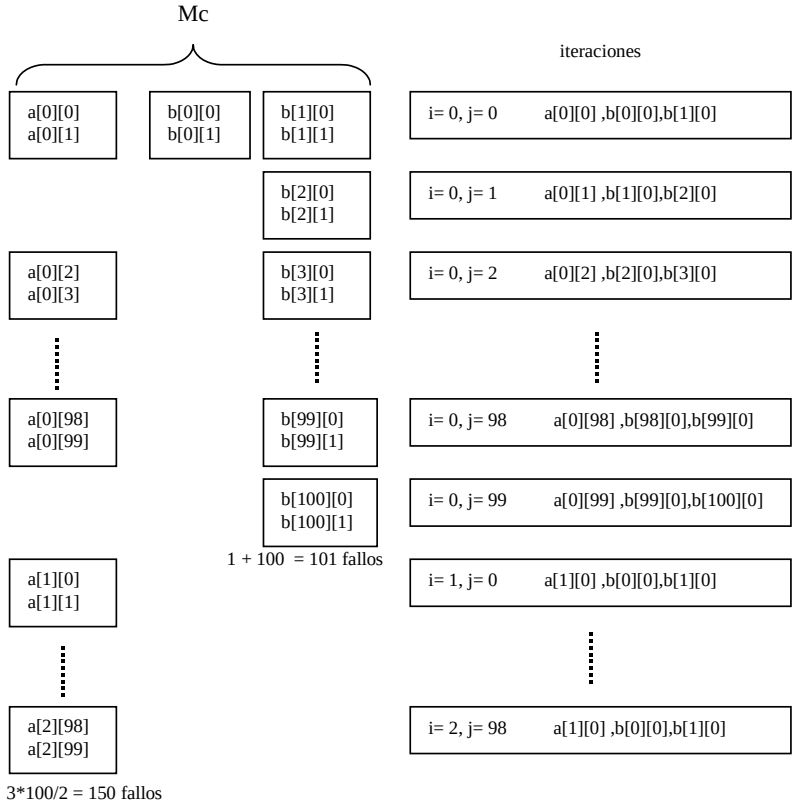
Cada elemento de los *arrays a[i][j]* y *b[i][j]* ocupan 8 bytes (doble precisión) y están dispuestos en memoria en orden ascendente de sus índices, es decir:

a[0][0]	b[0][0]
a[0][1]	b[0][1]
a[0][2]	b[0][2]

.....	.....
a[0][99]	b[0][99]
a[1][0]	b[1][0]
a[1][1]	b[1][1]
a[1][2]	b[1][2]
.....	.....
a[1][99]	b[1][99]
a[2][0]	b[2][0]
a[2][1]	b[2][1]
a[2][2]	b[2][2]
.....	.....
a[2][99]	b[2][99]

Cada 2 elementos consecutivos ocupan un bloque, por lo que *a[ ][ ]* se beneficiará de la localidad espacial: los valores pares de *j* producirán fallos (forzosos: primera lectura) y los impares aciertos (puesto que se llevan a Mc en los mismos bloques que los pares). Por tanto, teniendo en cuenta que tenemos 3 filas y 100 columnas, el número de fallos será *(3 * 100)/2 = 150*.

El *array b[ ][ ]* no se beneficia de la localidad espacial ya que los accesos no se realizan en el orden en que están almacenados sus elementos. En cambio se beneficiará dos veces de la localidad temporal ya que se accede a los mismos elementos para cada iteración de *i*. Además, cada iteración de *j* usa los mismos valores de *b[ ][ ]* que la iteración anterior. Ignorando los posibles fallos por conflicto, el número de fallos en el acceso a *b* serán 101, es decir, 1 para *b[0][0]* y 100 para *b[1][0]* hasta *b[100][0]*. En la 1ª iteración se producirán dos fallos: *b[0][0]* y *b[1][0];* en la 2ª uno: *b[2][0],* puesto que *b[1][0]* ya está en Mc; en la última uno: *b[100][0].*



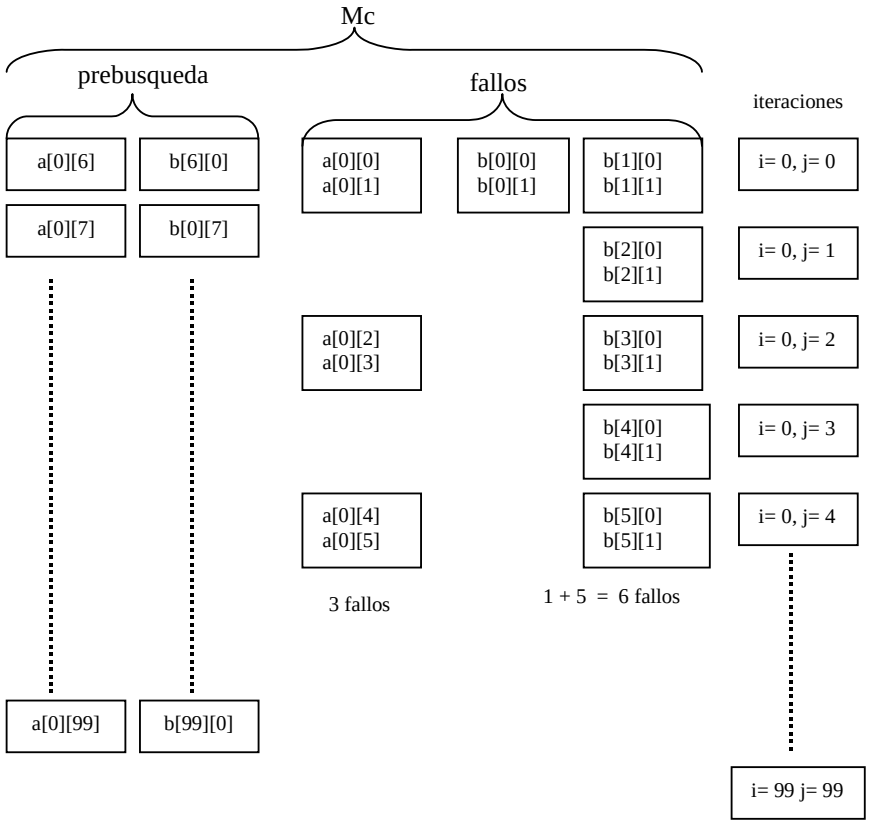
El número total de fallos será, pues, de 251.

Utilizando prebúsqueda el compilador puede transformar el programa en el siguiente:

```
for (j = 0; j < 100; j = j + 1)
    prefetch (b[j+6][0]);
    prefetch (a[0][j+6]);
            a[0][j] = b[j][0] * b[j+1][0];

for (i = 1; i < 3; i = i + 1)
    for (j = 0; j < 100; j = j + 1)
            prefetch (a[i][j+6]);
            a[i][j] = b[j][0] * b[j+1][0];
```

Se ha descompuesto el bucle en dos, el primero (iteración para i = 0) prebusca a partir de b[6][0] y a[0][6]:



Se ha elegido el valor 6 para el número de iteraciones que se buscan por adelantado.

En el segundo bucle sólo se prebusca a[1][6]... a[1][99], a[2][6]... a[2][99] puesto que todo el *array* b[i][j] ya ha sido prebuscado en el primer bucle y se halla en Mc. El número de fallos en las tres iteraciones de *a[][]* será 3*3 = 9. El número de fallos en la iteración de *b[][]* será 5+1 = 6.Luego en total solo se producen 15 fallos. El costo de evitar 236 fallos de cachés es ejecutar 400 instrucciones de prebúsqueda.

3.1.7. Optimizaciones del compilador

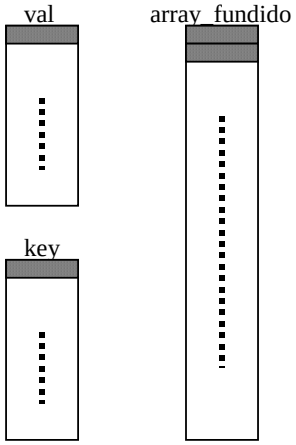
Las optimizaciones consisten en transformaciones del código fuente del programa, realizadas en tiempo de compilación, con el objetivo de aumentar la localidad espacial y/o temporal del programa, y consiguientemente reducir la tasa de fallos. Entre las transformaciones más importantes estudiaremos las siguientes:

3.1.7.1. Fusión de *arrays*

Se sustituyen varios *arrays* de igual tamaño por un único *array* de elementos estructurados. La transformación aumenta la localidad espacial si el programa referencia localmente las componentes de igual índice de los *arrays* originales.

Ejemplo:

programa original	programa transformado
int val[SIZE]; int key [SIZE];	struct merge { int val; int key;}; struct merge array_fundido[SIZE]



La transformación mejora la localidad espacial de los elementos de los dos arrays originales.

3.1.7.2. Fusión de bucles

programa original	programa transformado

for (i = 0; i < 100; i = i + 1)
for (j = 0; j < 100; j = j + 1)
a[i][j] = 2/b[i][j] *c[i][j];
 	
for (i = 0; i < 100; i = i + 1)
for (j = 0; j < 100; j = j + 1)
d[i][j] = a[i][j] + c[i][j];
	for (i = 0; i < 100; i = i + 1)
	for (j = 0; j < 100; j = j + 1)
	a[i][j] = 2/b[i][j] *c[i][j];
	d[i][j] = a[i][j] + c[i][j];

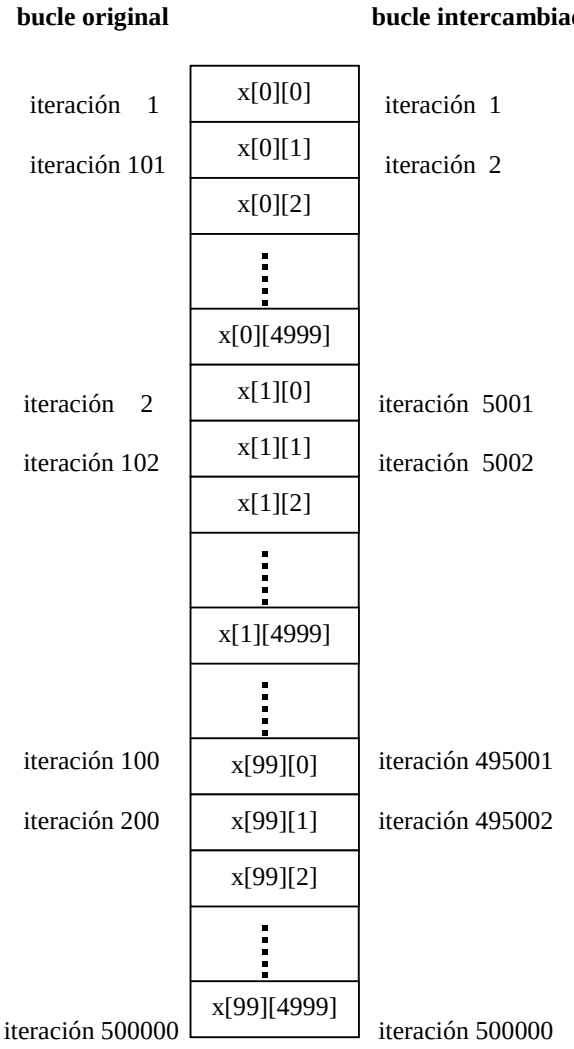
La transformación mejora la localidad temporal, ya que las referencias a *a[][]* y *c[][]* en el primer bucle del programa original se hacen separadas en el tiempo a las referencias a *a[][]* y *c[][]*

del segundo bucle. En cambio en el programa transformado estas referencias se hacen para los mismos valores de los índices en las 2 expresiones consecutivas.

3.1.7.3. Intercambio de bucles

programa original	programa transformado

for (j = 0; j < 100; j = j + 1)
for (i = 0; i < 5000; i = i + 1)
x[i][j] = 2*x[i][j];
	for (i = 0; i < 5000; i = i + 1)
	for (j = 0; j < 100; j = j + 1)
	x[i][j] = 2*x[i][j];



La transformación mejora la localidad espacial.

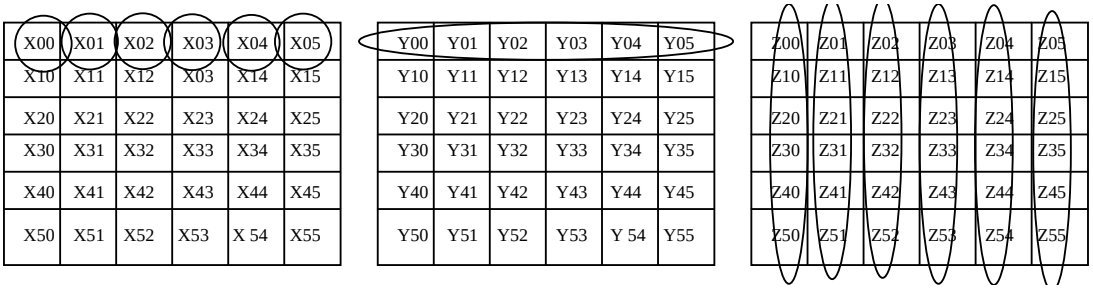
3.1.7.4. Descomposición en bloques

- Reduce la tasa de fallos aumentando la localidad temporal
- En lugar de operar sobre filas o columnas completas de un *array* los algoritmos de descomposición en bloques operan sobre submatrices o bloques
- El objetivo es maximizar los accesos a los datos cargados en la caché antes de ser reemplazados

-

Ejemplo: multiplicación de matrices

```
for (i = 1; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
            { r = 0;
              for (k = 0; k < N; k = k + 1){
                    r = r + Y[i][k] *Z[k][j];};
              X[i][j] = r;
              };
```

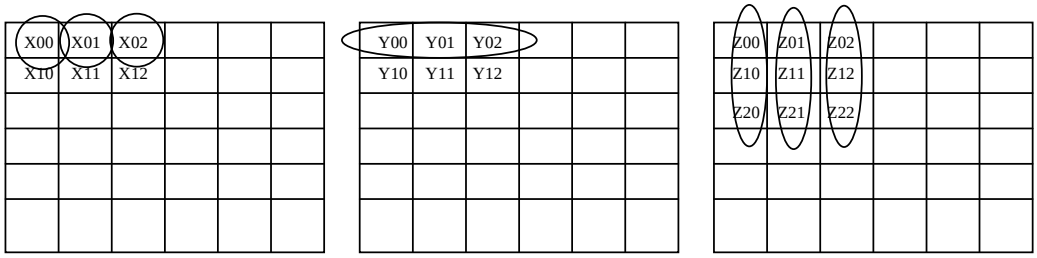


Como vemos, para calcular los X[ ][ ] de la primera fila vamos multiplicando la primera fila de Y[ ][ ] por cada una de las columnas de Z[ ][ ]. Los X[ ][ ] de la segunda fila se calculan de forma análoga, utilizando en este caso la segunda fila de Y[ ][ ]. Y así sucesivamente. La matriz Z[ ][ ] debería permanecer en la caché durante el cálculo de todos los elementos de X[ ][ ]. Sin embargo, si las matrices son de gran tamaño esto no será posible y se producirán una serie de fallos de caché que llevarán sucesivas veces los elementos de Z[ ][ ] a la caché. Para evitar esto podemos transformar el programa de la siguiente forma:

```
for (jj = 0; jj < N; jj = jj + B)
for (kk = 0; kk < N; kk = kk + B)
for (i = 0; i < N; i = i + 1)
      for (j = jj; j < min(jj + B, N); j = j + 1)
            {  r = 0;
               for (k = kk; k  <  min(kk + B,N); k = k + 1)
                  { r = r + Y[i][k] *Z[k][j];};
              X[i][j] = X[i][j] +r;
            };
```

Este programa va calculando parcialmente los valores de x[ ][ ] para que los bloques de elementos de Y[ ][ ] y Z[ ][ ] sean utilizados totalmente cuando se llevan a la caché, aumentando su localidad temporal. Los índices *jj* y *kk* van determinando los bloques de tamaño *B*. Los índices *j* y *k* iteran sobre cada uno de los bloques calculando productos parciales de tamaño B que se van

acumulando sobre la variable r. Los valores de r se acumulan sobre el X[ ][ ] final dentro del bucle correspondiente al índice *i*.



3.2. Reducción de la penalización de fallos

3.2.1. Prioridad a los fallos de lectura frente a los de escritura

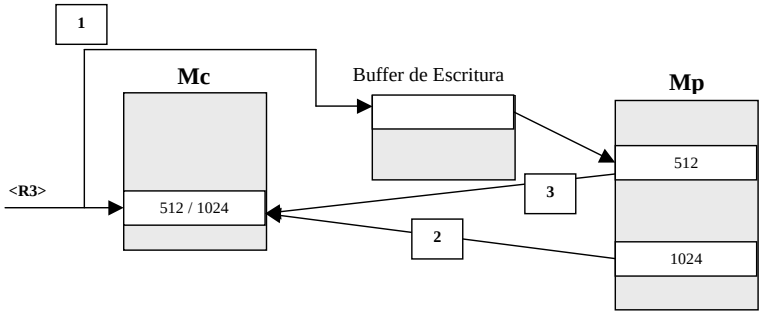
Como vimos con anterioridad, con una caché de escritura directa (*writre throuhg*) la mejora de rendimiento se consigue utilizando un buffer de escritura de tamaño apropiado. Sin embargo, esto complica los accesos a memoria debido a que el buffer en un momento determinado puede contener aún el valor actualizado (escrito) de una posición que debería estar en Mp para servir un fallo de lectura.

Ejemplo: Supongamos que una Mc de correspondencia directa y escritura directa (*writre throuhg*) hace corresponder los bloques en los que se encuentran las direcciones 512 y 1024 sobre la misma línea de Mc. Supongamos que se ejecuta el siguiente programa:

(1) M[512] <-- <R3> // escritura sobre M[512]

(2) R1 <-- M[1024] // lectura de M[1024]

(3) R2 <-- M[512] // lectura de M[512]



Cuando se ejecuta (1) se lleva <R3> al buffer de escritura y a la línea de Mc a la que pertenece la posición 512 (supuesto acierto de escritura, es decir, el bloque al que pertenece 512 se encuentra en Mc).

Cuando se ejecuta (2) se produce un fallo de lectura y se sustituye en Mc el bloque al que pertenece 512 por el bloque al que pertenece 1024

Cuando se ejecuta (3) se vuelve a producir un fallo de lectura para llevar de nuevo el bloque al que pertenece 512 a Mc, pero es posible que éste no esté actualizado por el efecto de (1) si aún el contenido del buffer no se ha escrito en Mp

Este problema de inconsistencia se puede resolver de dos maneras:

1) Retrasando el servicio del fallo de lectura producido por (3) hasta que el buffer esté vacío (todo su contenido se haya reflejado en Mp). Esta solución penaliza bastante los fallos de lectura, pues los datos empíricos demuestran que la espera sistemática a que el buffer se vacíe para servir

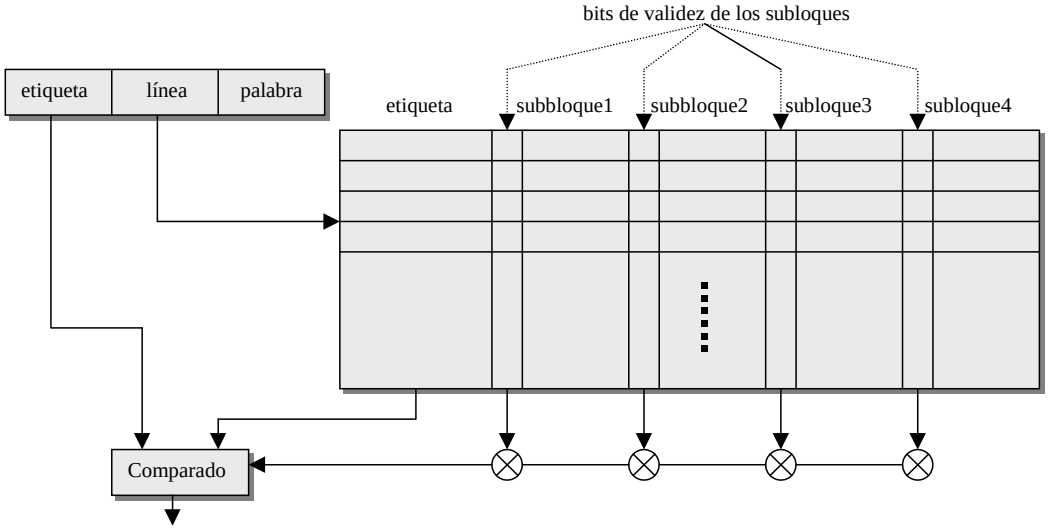
un fallo de lectura puede penalizar estos fallos por un factor de 1.5. Por ello es recomendable dar prioridad a la lectura (es mucho más frecuente que la escritura) adoptando la segunda alternativa.

2) Incorporar al proceso asociado al fallo de lectura producido por (3) la comprobación de si el buffer contiene la posición 512, y continuar si el resultado es falso.

3.2.2. Utilización de sub-bloques dentro de un bloque

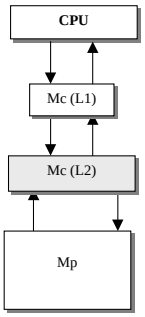
La utilización de bloques de gran tamaño no solo disminuyen la tasa de fallos sino que reduce el espacio de memoria caché dedicado al directorio (almacenamiento de las etiquetas de los bloques residentes en una línea). Sin embargo, bloques grandes aumentan la penalización por fallos debido al aumento de la cantidad de datos que se deben transferir en el servicio de cada fallo. Una forma de disminuir la penalización por fallos sin modificar los otros factores positivos consiste en dividir el bloque en sub-bloques y asociar un bit de validez a cada sub-bloque. De esta forma, cuando se produzca un fallo, no será necesario actualizar más que el sub-bloque que contiene la palabra referenciada. Esta alternativa hace que no sólo haya que comprobar si el bloque está en la caché comparando etiquetas, sino que habrá que asegurar que el sub-bloque que contiene la palabra referenciada es un sub-bloque válido, es decir, con datos actualizados.

También podemos ver esta alternativa de diseño como una forma de economizar información de directorio asociando una sola etiqueta a un grupo de bloques, e indicando con un bit particular asociado a cada bloque (bit de validez) su estado de actualización cuando el grupo está en Mc. En la siguiente figura se muestra un esquema de esta alternativa de diseño:



3.2.3. Utilización de un segundo nivel de caché

Como los fallos se sirven leyendo bloques de Mp, una alternativa para disminuir la penalización por fallo consiste en disminuir el tiempo de acceso a Mp utilizando el mismo mecanismo caché, es decir, utilizando una caché intermedia o de segundo nivel (L2) entre Mc (L1) y Mp.



$Tiempo\_acceso\_medio = Tiempo\_acierto_{N1} + Tasa\_fallos_{N1} * Penalización\_fallos_{N1}$

$Penalización\_fallos_{N1} = Tiempo\_acierto_{N2} + Tasa\_fallos_{N2} * Penalización\_fallos_{N2}$

Cuando tenemos varios niveles de cachés hay que diferenciar entre la tasa de fallos local y la global:

Tasa_fallos_local = nº de fallos / nº de accesos a la caché

Tasa_fallos_global = nº de fallos / nº total de accesos realizados por la CPU

En general se cumple:

$Tasa\_fallos\_local \geq Tasa\_fallos\_global$

Y en particular:

$Tasa\_fallos\_local_{N1} = Tasa\_fallos\_global_{N1}$

$Tasa\_fallos\_local_{N2} > Tasa\_fallos\_global_{N2}$

Ejemplo:

1000 referencias a un sistema de cahé de dos niveles

40 fallos se producen en N1

20 fallos se producen en N2

$Tasa\_fallos\_local_{N1} = Tasa\_fallos\_global_{N1} = 40/1000 = 0.04\ (4\%)$

$Tasa\_fallos\_local_{N2} = 20/40 = 0.5\ (50\%)$

$Tasa\_fallos\_global_{N2} = 20/1000 = 0.02\ (2\%)$

3.3. Reducción del tiempo de acierto

El tiempo de acierto podemos optimizarlo (minimizarlo) actuando sobre tres factores:

3.3.1. Cachés pequeñas y simples

- El hardware pequeño acelera la comparación de etiquetas y por tanto el tiempo de acierto
- También hace posible su incorporación al chip de la CPU, eliminando el conexionado externo y por tanto el tiempo de acceso desde la CPU

3.3.2. Evitar traducción de direcciones durante la indexación de las cachés

- Utilización de direcciones virtuales en las cachés

3.3.3. Escrituras segmentadas para rápidos aciertos de escritura

Los aciertos de lectura son más rápidos que los de escritura, entre otros motivos porque en los primeros se puede leer el dato de Mc al tiempo que se comprueba si su etiqueta coincide con la de la dirección física. Si no coincide se ignora el dato leído. Esto no es posible en las escrituras, pero sí podemos simultanear la escritura de un dato con la comparación de la etiqueta del siguiente. Es decir, segmentar (*pipe-line*) la escritura sobre Mc. De esta forma se aceleran los aciertos de escritura.