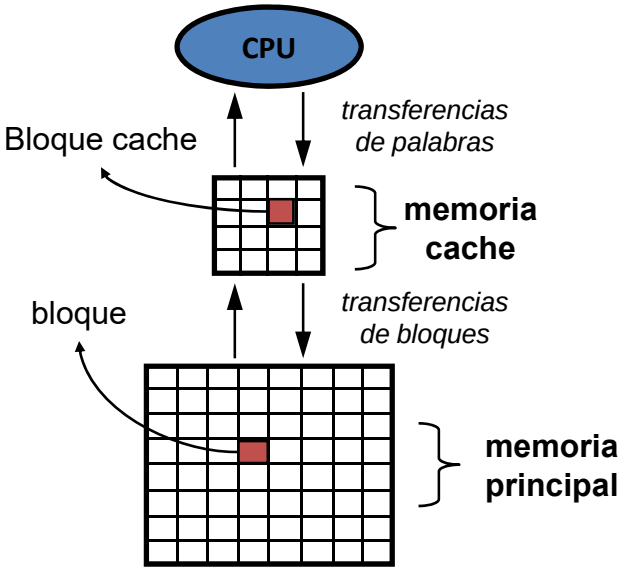
Tema 6. Memoria caché

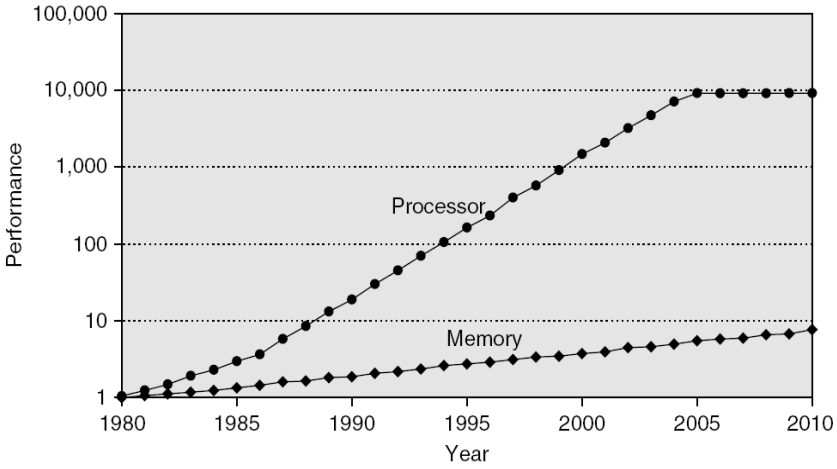
Índice

1. Introducción

➢ Motivación de la memoria cache

- La velocidad de la CPU ha ido aumentado a un ritmo superior al que lo hacía la memoria principal.
- En la siguiente gráfica se muestra la diferencia progresiva a lo largo del tiempo entre la capacidad de la CPU para ejecutar instrucciones y la de la memoria principal (tecnología DRAM) para suministrarlas.
- La memoria caché es una solución estructural a este problema de diferencia de velocidades, y consiste en interponer una memoria más rápida (tecnología SRAM) y más pequeña entre la principal y la CPU.
- Un mecanismo específico fundamentado en la localidad de referencia de los programas hace posible que las palabras de memoria se encuentren en la caché cuando son referenciadas por la CPU.





Tipo de memoria	Tiempo de acceso (ns)
SRAM	0.5-2.5
DRAM	50-70

1. Introducción

➢ Localidad de referencia de los programas

- Los programas manifiestan una propiedad denominada ***localidad de referencias****: tienden a reutilizar los datos e instrucciones que utilizaron recientemente o están próximos a los utilizados recientemente.*

- Esta propiedad se explota en el diseño del sistema de gestión de memoria en general y de la memoria caché en particular.

- La localidad de referencia se manifiesta en una **doble dimensión**: temporal y espacial.

  - **<u>Localidad temporal</u>:**

  Las palabras de memoria accedidas recientemente tienen una alta probabilidad de volver a ser accedidas en el futuro cercano.

  - Motivada por la existencia de bucles, subrutinas y accesos a la pila de variables locales.

  - **<u>Localidad espacial</u>**:

  Las palabras próximas a las recientemente referenciadas tienen una alta probabilidad de ser referenciadas en el futuro cercano.

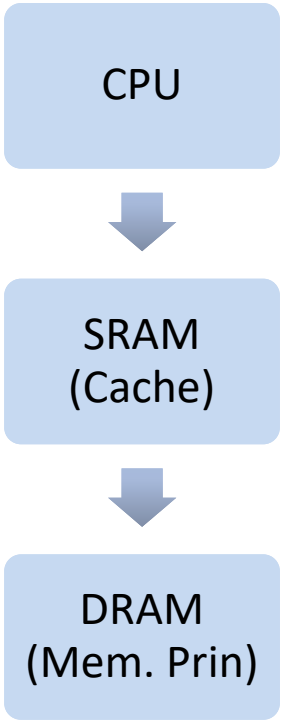
  - Viene motivada fundamentalmente por la linealidad de los programas y el acceso a las estructuras de datos regulares (arrays)

- 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.

1. Introducción

¿Cómo se explota la localidad temporal?

- Cada vez que se accede a memoria principal se copia la palabra accedida en la memoria caché, más pequeña y rápida (SRAM).
  - Esta copia ocasiona una penalización en el primer acceso
  - Penalización que será amortizada con creces porque los sucesivos accesos a esa palabra (localidad temporal) se realizarán sobre la memoria caché, uno o dos órdenes de magnitud más rápida



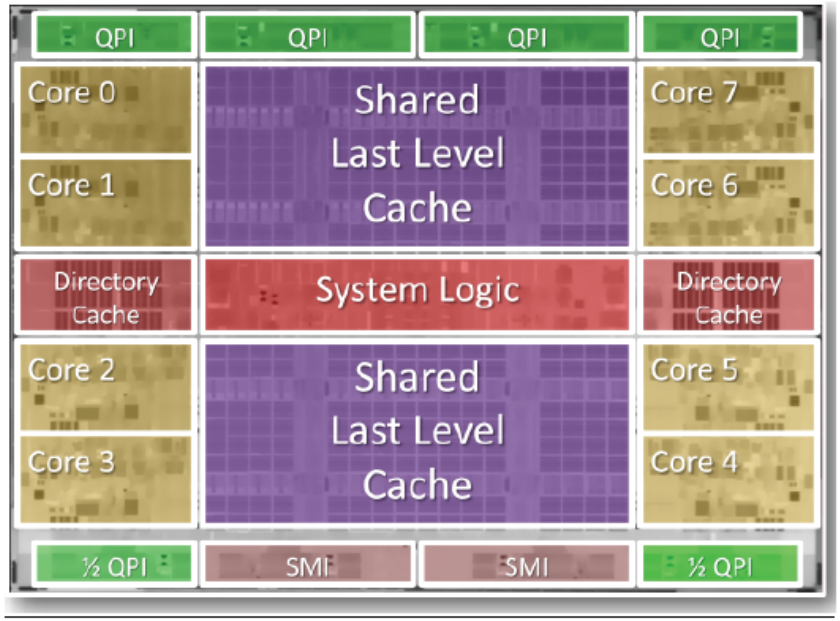
¿Cómo se explota la localidad espacial?

- Cada vez que se accede a una dirección de memoria se copia en la caché la palabra accedida y las palabras próximas (bloque).
  - Por ejemplo, si se accede al elemento a[1], se lleva a la cache a[0],a[1],a[2] y a[3].
  - De nuevo hay una penalización inicial que será amortizada si la CPU accede a las restantes palabras del bloque (localidad espacial), accesos que se harán sobre la memoria caché.

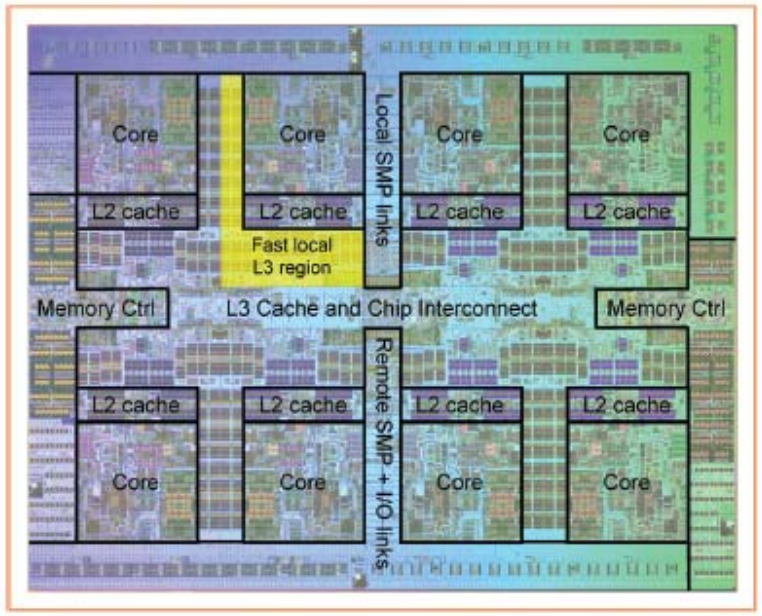
1. Introducción

➢Tamaño físico de la cache

- Ocupa del 50 al 75 % del área
- Más del 80% de los transistores
- Ejemplos:



Intel Poulson 38 MB total
L1 datos 16KB, L1 instrucciones 16KB
L2 datos 256KB, L2 instrucciones 512KB
L3 32MB



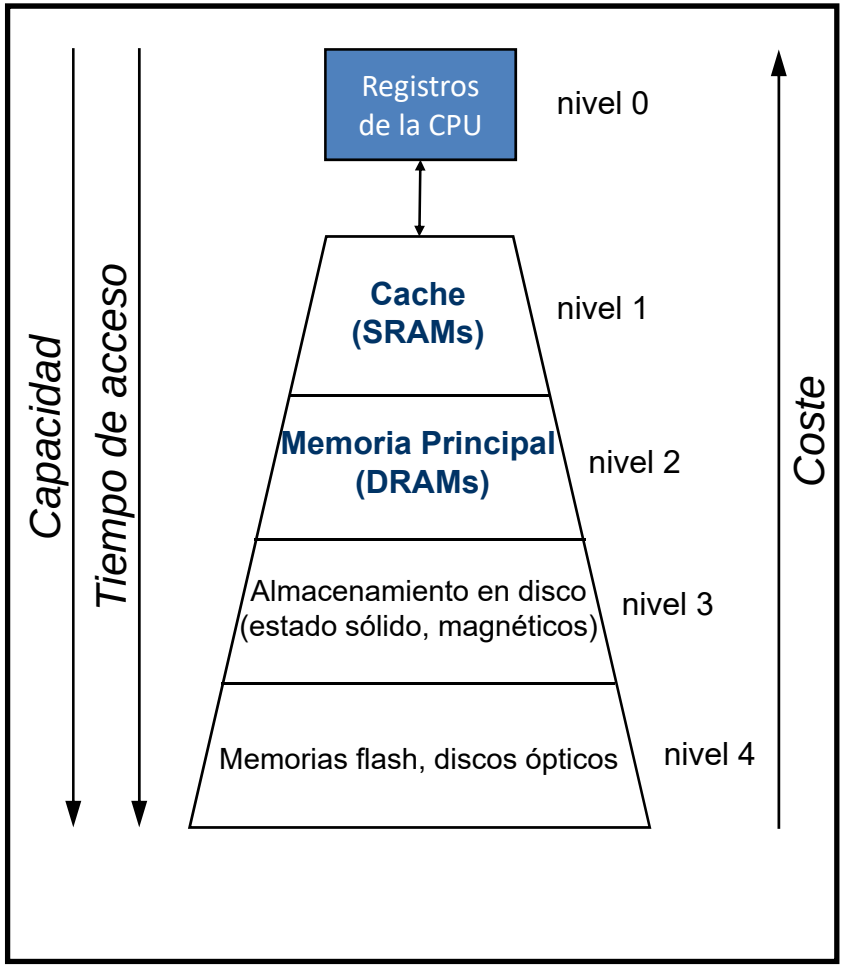
IBM Power 7 total 34,5 MB
L1 datos 32KB, L1 instrucciones 32KB
L2 256KB
L3 32MB

2. Jerarquía de memoria

Jerarquía de memoria

- En un computador típico existen diversos tipos de memoria, *organizados de forma jerárquica*:
  - Registros de la CPU
  - Memoria Cache
  - Memoria Principal
  - Memoria Secundaria (discos)
  - Memorias flash y CD-ROMs

- Propiedades de la jerarquía:
  1. **Inclusión**: la información almacenada en el nivel Mi debe encontrarse también en los niveles Mi+1, Mi+2, …, Mn
  2. **Coherencia**: si un bloque de información se modifica en el nivel Mi, deben actualizarse los niveles Mi+1,.., Mn



2. Jerarquía de memoria

➢ Objetivo de la gestión de la jerarquía de memoria

- Optimizar el uso de la memoria
- Hacer que el usuario tenga la ilusión de que dispone de una memoria con:
  - ⇨*Tiempo de acceso similar al del sistema más rápido*
  - ⇨*Coste por bit similar al del sistema más barato*
- Para la mayor parte de los accesos a un bloque de información, este bloque debe encontrarse en los niveles bajos de la jerarquía de memoria

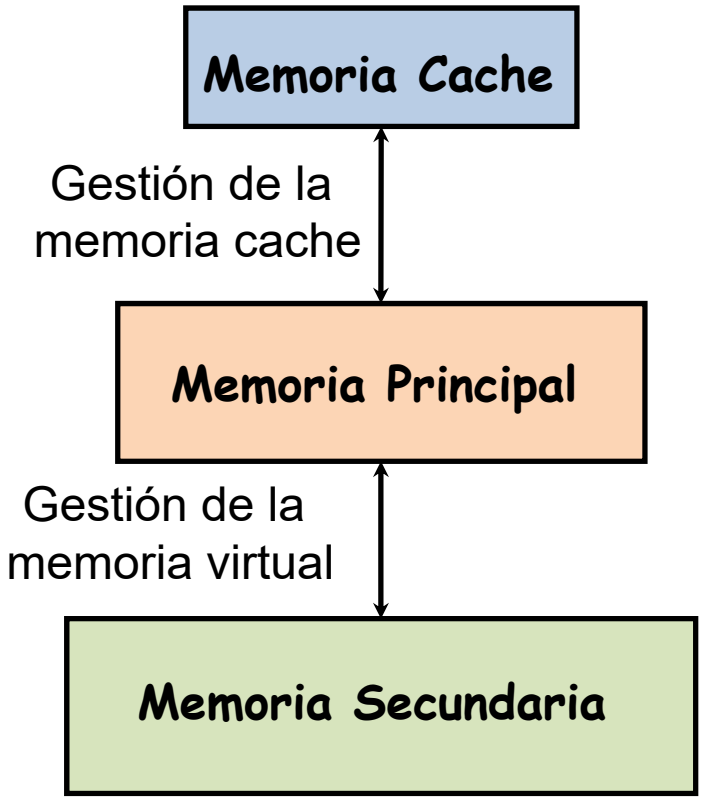
➢ Niveles a los que afecta la gestión de la jerarquía memoria

- Se refiere a la gestión dinámica, en tiempo de ejecución de la jerarquía de memoria
- Esta gestión de la memoria sólo afecta a los niveles 1 (cache), 2 (mem. principal) y 3 (mem. secund.)
  - ⇨ El nivel 0 (registros) lo asigna el compilador en tiempo de compilación
  - ⇨ El nivel 4 (cintas y discos ópticos) se utiliza para copias de seguridad (back-up)

2. Jerarquía de memoria

➢ Gestión de la jerarquía de memoria

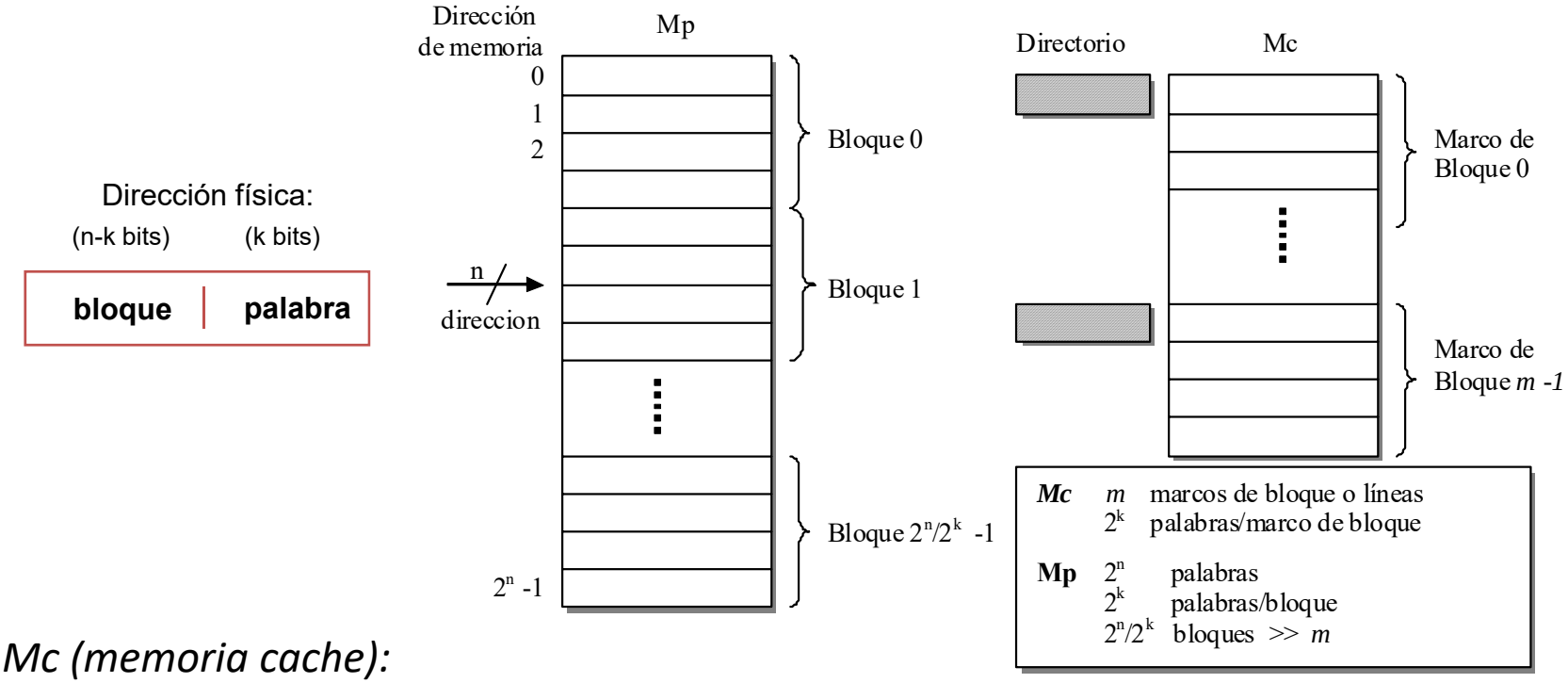
- **Gestión de la memoria cache**
  - Controla la transferencia de información entre la memoria cache y la memoria principal
  - Suele llevarse a cabo mediante Hardware específico (MMU o "Management Memory Unit")
- **Gestión de la memoria virtual**
  - Controla la transferencia de información entre la memoria principal y la memoria secundaria
  - Parte de esta gestión se realiza mediante hardware específico (MMU) y otra parte la realiza el S.O



3. Principios de funcionamiento de la memoria caché

➢ Estructura del sistema memoria cache/principal

- *Mp (memoria principal):*
  - Constituida por 2^n palabras "dividida" en 2^{n-k} bloques de 2^k palabras cada uno

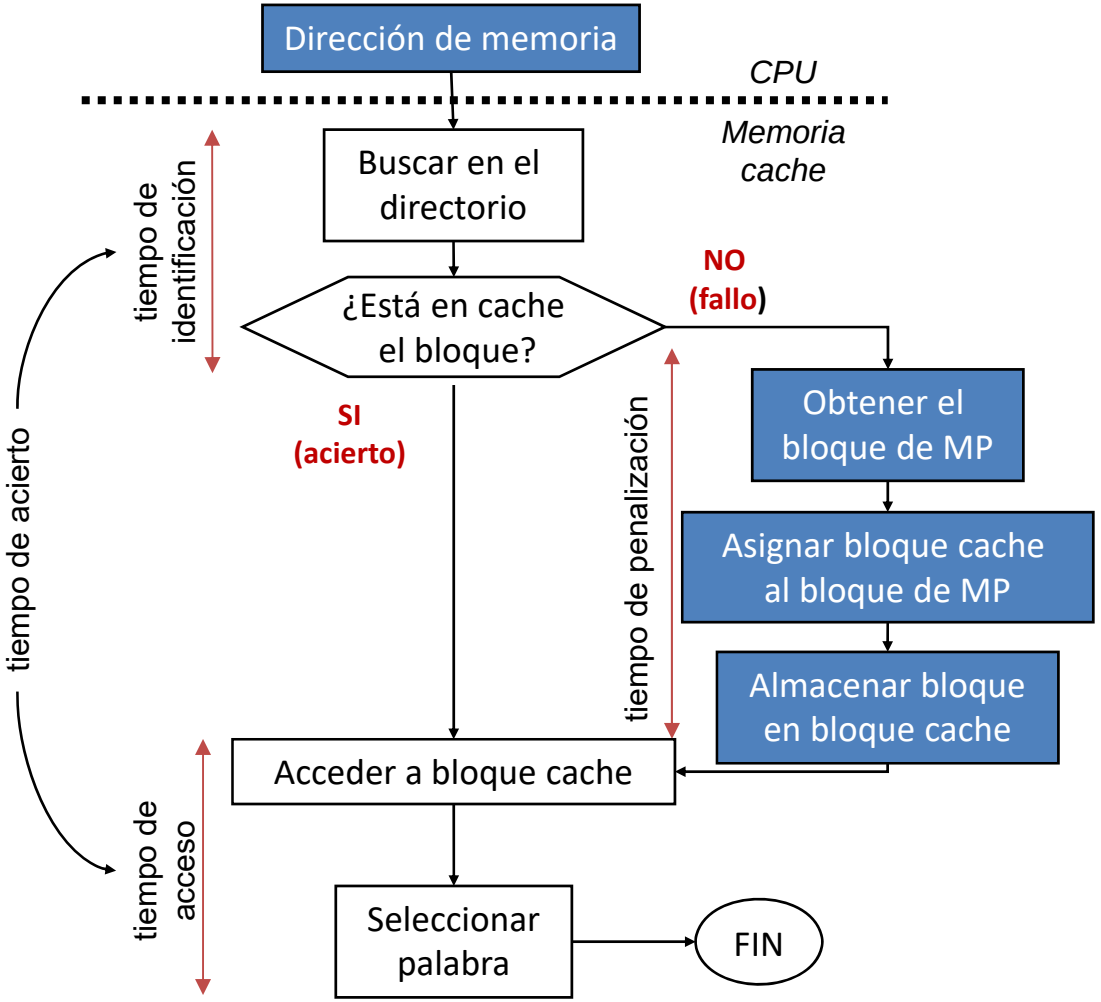


- *Mc (memoria cache):*
  - Constituida por *m* marcos de bloque (o líneas) de 2^k palabras
- *Directorio (en memoria cache):*
  - Sirve para determinar el bloque de Mp alojado en cada línea de Mc.

3. Principios de funcionamiento de la memoria caché

➢ Funcionamiento general de la memoria caché

- Cuando la CPU genera una referencia, busca en la cache
  - Si la referencia no se encuentra en la cache: **FALLO**
  - Cuando se produce un fallo, no solo se transfiere una palabra, sino que se lleva un **BLOQUE** completo de información de la Mp a la Mc
  - *Por la propiedad de localidad temporal*
    - Es probable que en una próxima referencia se direccione la misma posición de memoria
    - Esta segunda referencia no producirá fallo: producirá un **ACIERTO**
  - *Por la propiedad de localidad espacial*
    - Es probable que próximas referencias sean direcciones que pertenecen al mismo bloque
    - Estas referencias no producen fallo, producirán **ACIERTO**



4. Elementos de diseño

➢ Alternativas de diseño en una memoria caché

▪ **Función de correspondencia (emplazamiento):** 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 únicas o independientes:** para datos e instrucciones

▪ **Parámetros de rendimiento**

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

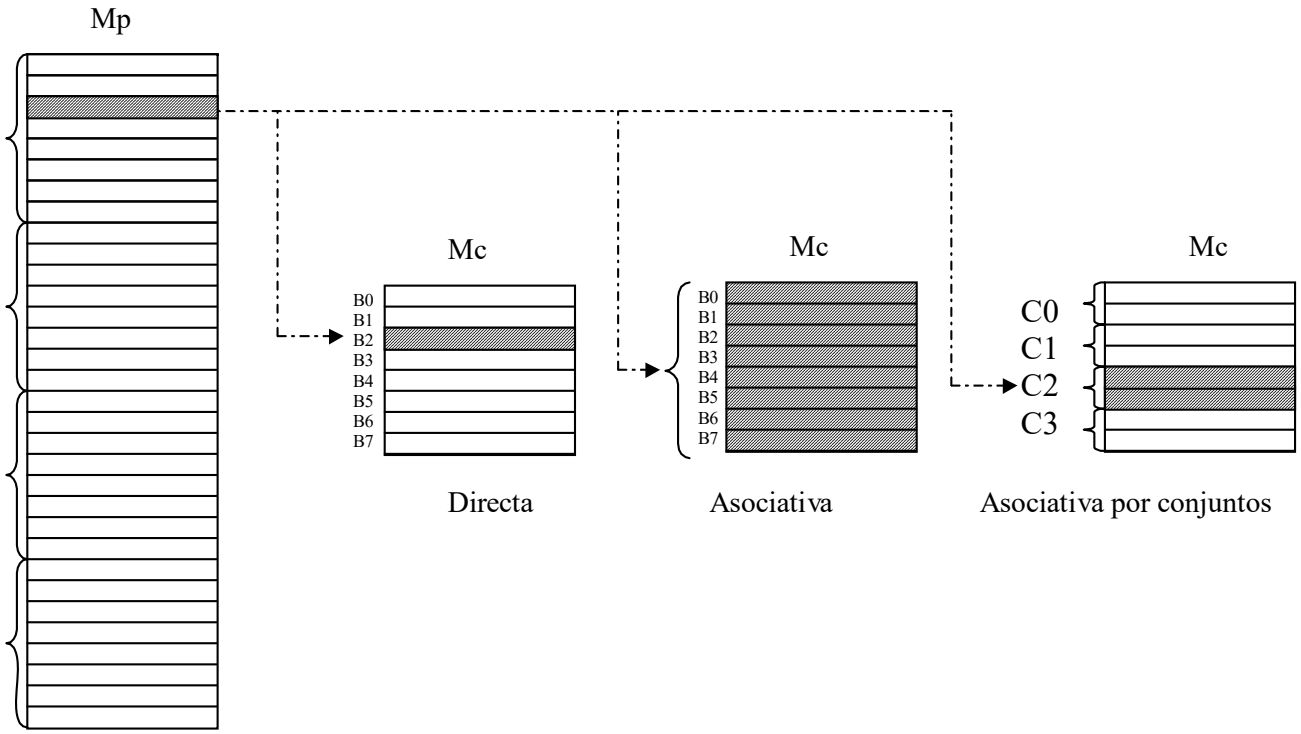
  Nr = número de referencias a memoria
  Na = número de aciertos caché
  Nf = número de fallos

- Evidentemente se cumple: $Ta = 1 – Tf$

4. Elementos de diseño

➢ Función de correspondencia

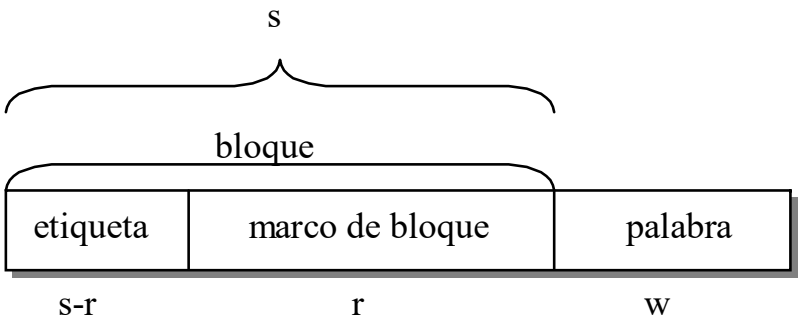
- Determina las posibles líneas de la caché (marcos de bloque) en las que se puede ubicar un determinado bloque de la memoria principal.
- Existen tres funciones de correspondencia: *directa*, *asociativa* y *asociativa por conjuntos*.
  - **Directa**: un bloque de Mp sólo puede ubicarse en una línea de la caché, aquella que coincide con el bloque cuando superponemos Mc sobre Mp respetando fronteras de Mc
  - **Asociativa**: un bloque de Mp puede ubicarse en cualquier línea de Mc.
  - **Asociativa por conjuntos**: es un compromiso entre las dos anteriores.



4. Elementos de diseño

➢ Correspondencia directa (1)

- El bloque *Bj* de *Mp* se puede ubicar sólo en el marco de bloque *MBi* tal que ***i = j mod m,*** con *m* = número total de líneas que tiene la caché.



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.

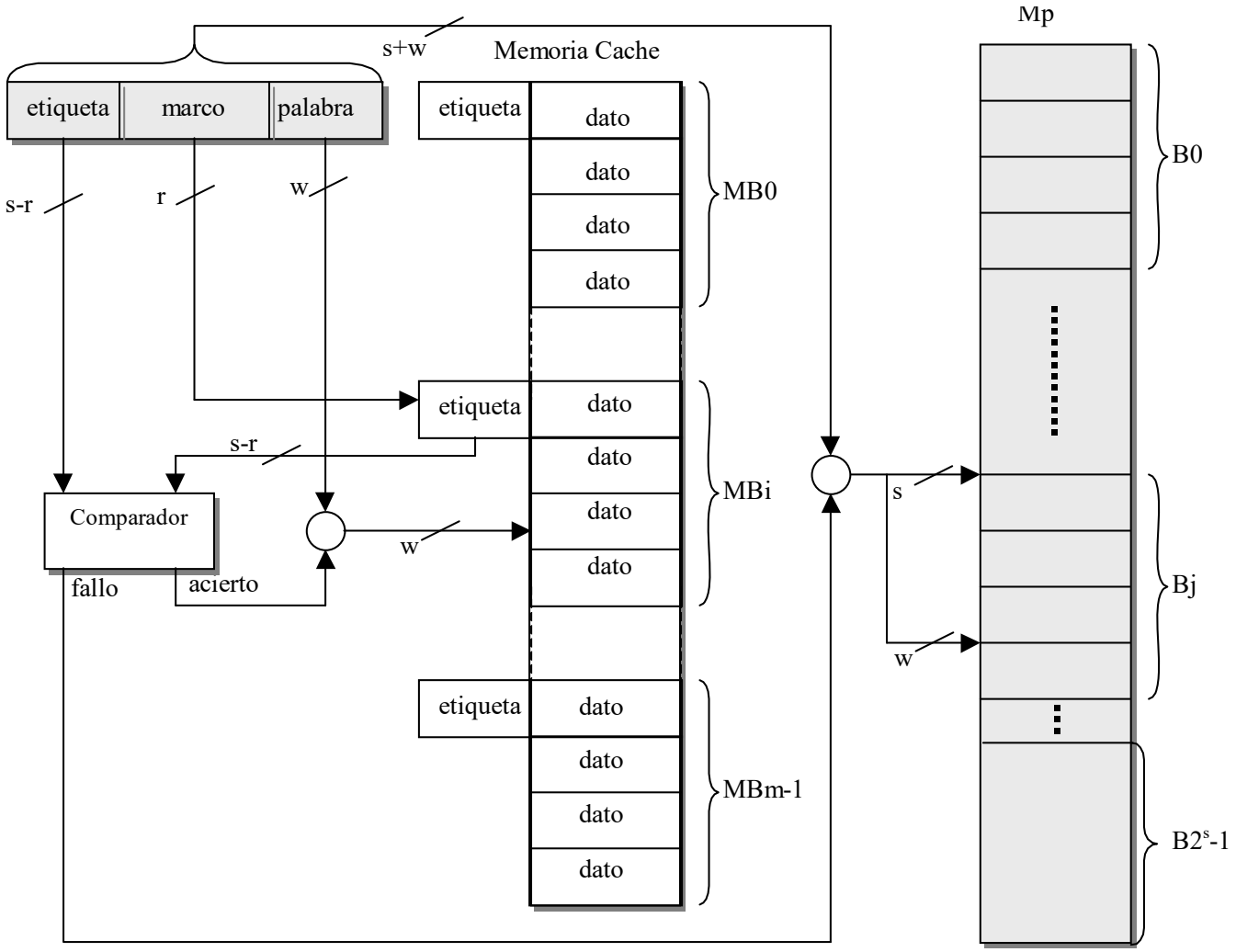
- El directorio caché en correspondencia directa contendrá un registro de *s* - *r* bits por cada marco de bloque para contener la etiqueta del bloque ubicado en ese momento en dicho marco.

4. Elementos de diseño

➢ Correspondencia directa (2)

- El esquema de acceso a una Mc con correspondencia directa es el siguiente:

Los bits del campo *marco* (o *línea*) de la dirección determinan la línea leída.
- Si la etiqueta de la dirección coincide con la etiqueta contenida en la línea, hay acierto, es decir, el bloque contenido en la línea leída es el mismo al que pertenece la dirección que accede al sistema de memoria.
- En caso contrario se produce un fallo de caché.



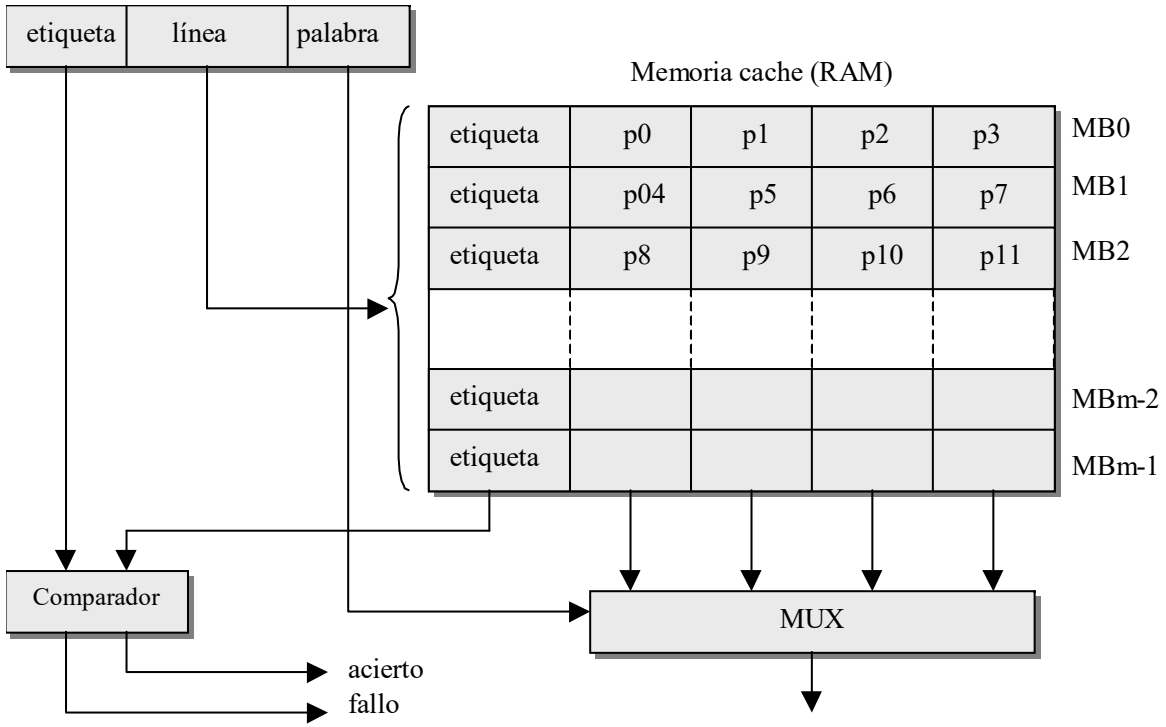
4. Elementos de diseño

➢ Correspondencia directa (3)

- Desde el punto de vista del diseño una Mc con correspondencia directa se organiza como una memoria RAM con longitud de palabra suficiente para contener una línea y los bits de etiqueta.
- El número de palabras lo determinará el número de líneas.
- El esquema de acceso es el siguiente:

Con los bits de *línea* se accede a una palabra de la RAM, cuyos bits más significativos contienen la etiqueta, que se compara con el campo etiqueta de la dirección.

Los bits del campo palabra seleccionan la palabra específica del bloque

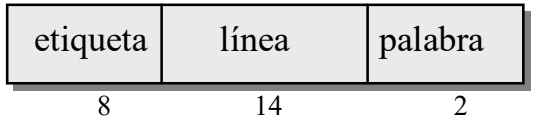


4. Elementos de diseño

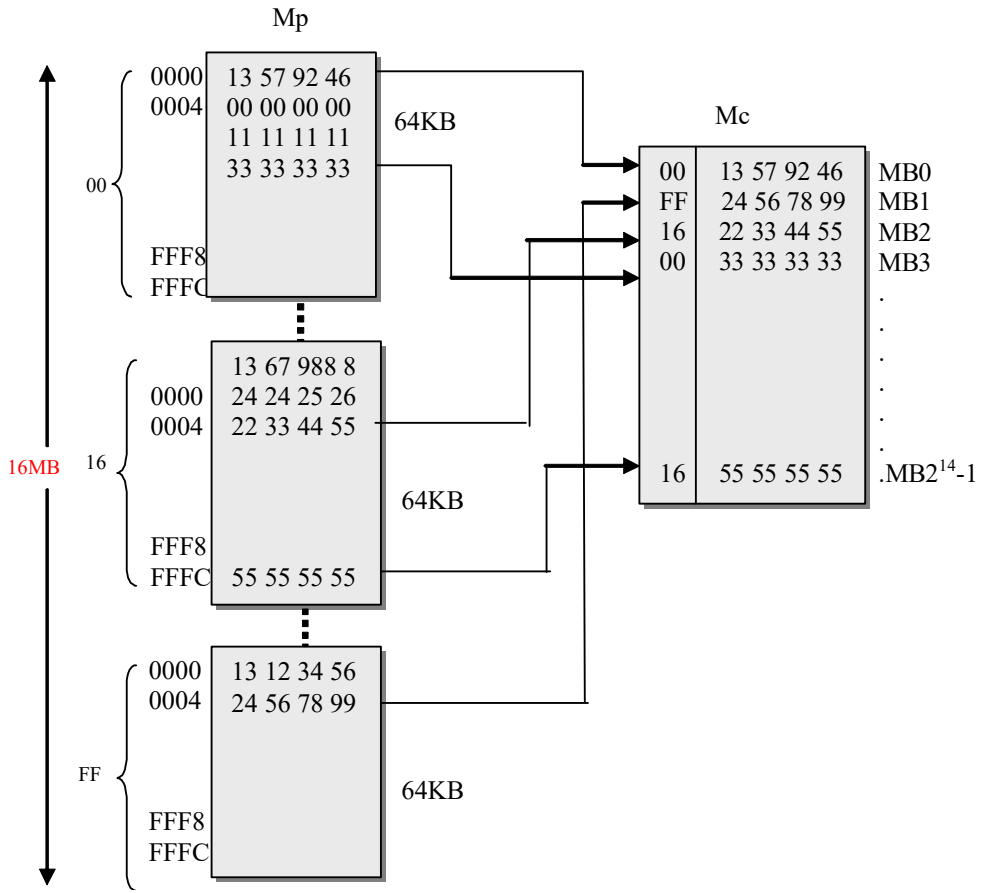
➢ Correspondencia directa (4)

- **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



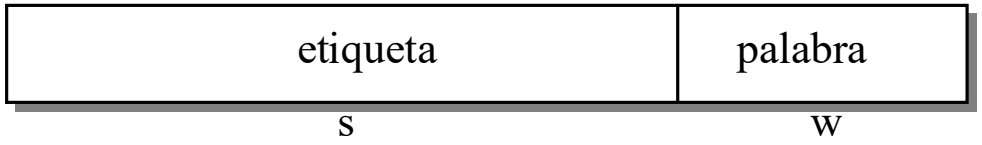
El dibujo de la derecha muestra un posible estado de Mc y Mp con correspondencia directa.
Mp se ha contemplado dividida en zonas de tamaño igual a Mc para facilitar gráficamente la correspondencia entre líneas de Mc y bloques de Mp



4. Elementos de diseño

➢ Correspondencia asociativa (1)

- Un bloque *Bj* de Mp se puede ubicar en cualquier marco de bloque de *Mc*.



2^s bloques de Mp

2^r marcos de bloque de Mc

- La etiqueta tiene *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.

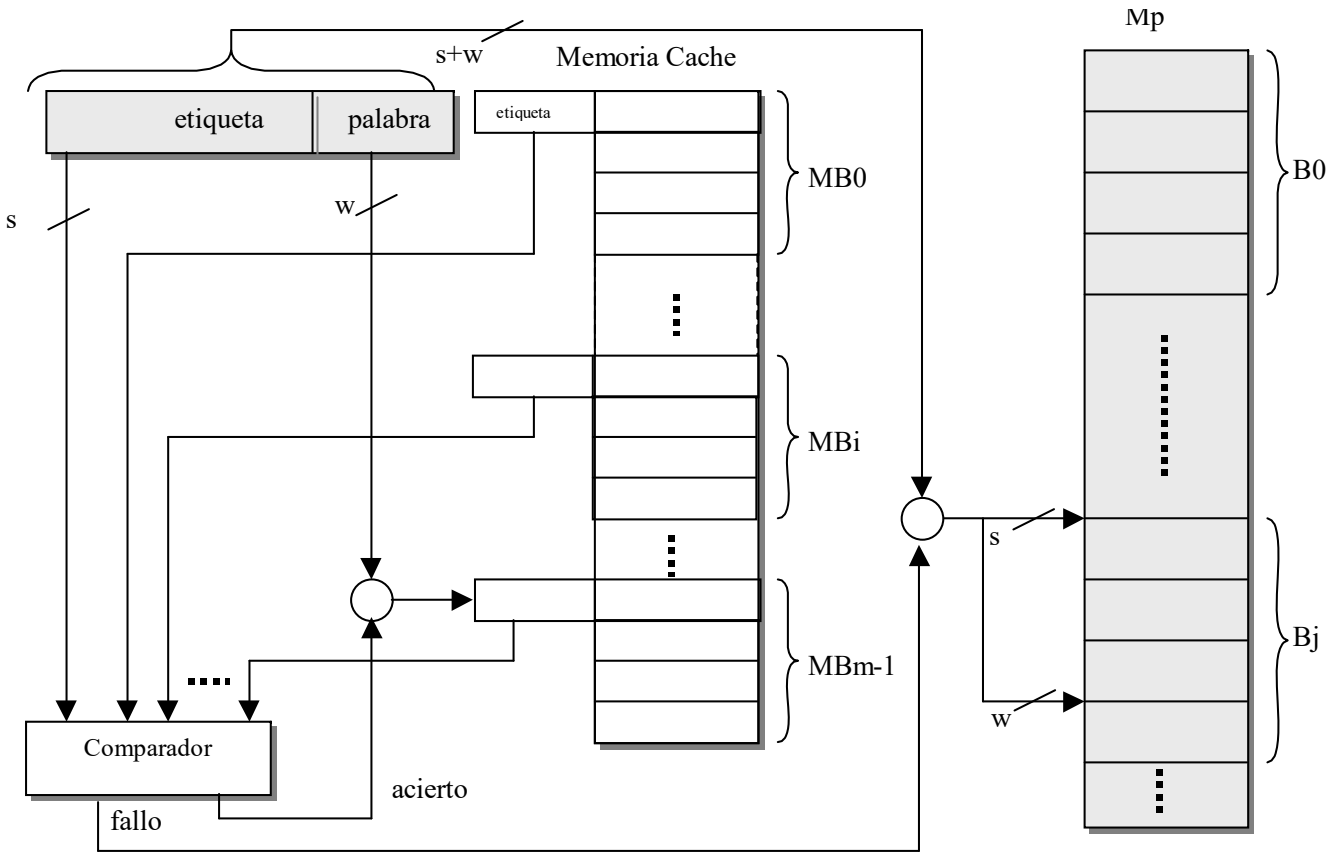
- El directorio caché en correspondencia asociativa contendrá, pues, un registro de *s* bits por cada marco de bloque para contener la etiqueta del bloque ubicado en ese momento en dicho marco

4. Elementos de diseño

➢ Correspondencia asociativa (2)

- El esquema de acceso a una Mc con correspondencia asociativa es el siguiente:

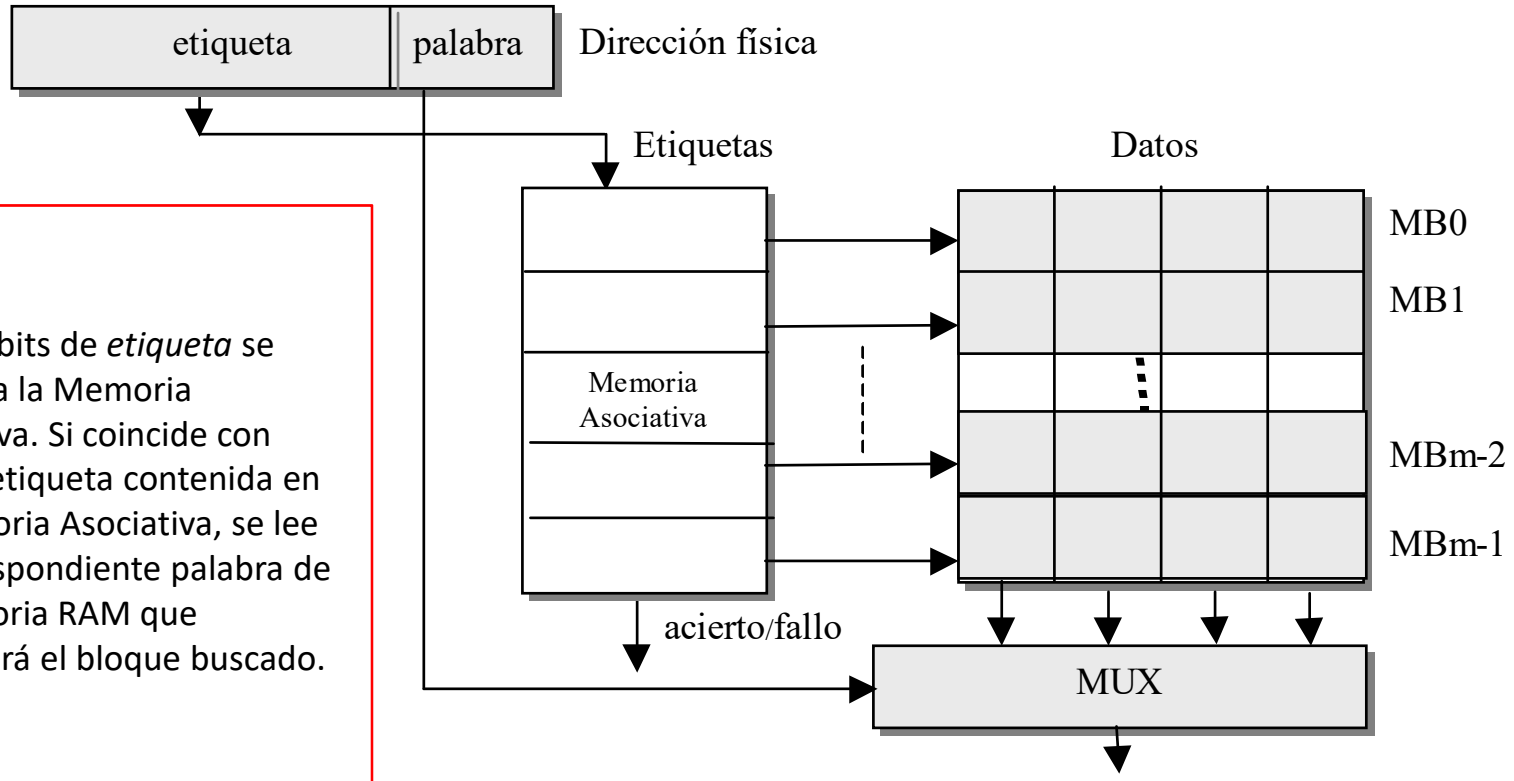
En este caso se compara el campo etiqueta de la dirección con las etiquetas de todas las líneas de Mc.



4. Elementos de diseño

➢ Correspondencia asociativa (3)

- Desde el punto de vista del diseño una Mc con correspondencia asociativa se organiza con una memoria RAM con longitud de palabra suficiente para contener una línea, y una Memoria Asociativa para contener las etiquetas de los bloques actualmente en Mc.
- El esquema de acceso es el siguiente:

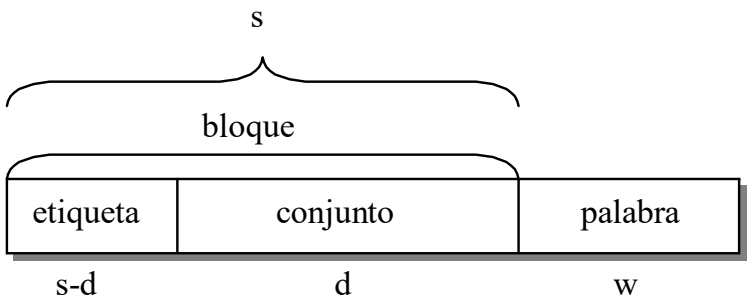


Con los bits de *etiqueta* se accede a la Memoria Asociativa. Si coincide con alguna etiqueta contenida en la Memoria Asociativa, se lee la correspondiente palabra de la memoria RAM que contendrá el bloque buscado.

4. Elementos de diseño

➢ Correspondencia asociativa por conjuntos (1)

- Las líneas de ***Mc* se dividen 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 ***i = j mod v*.**



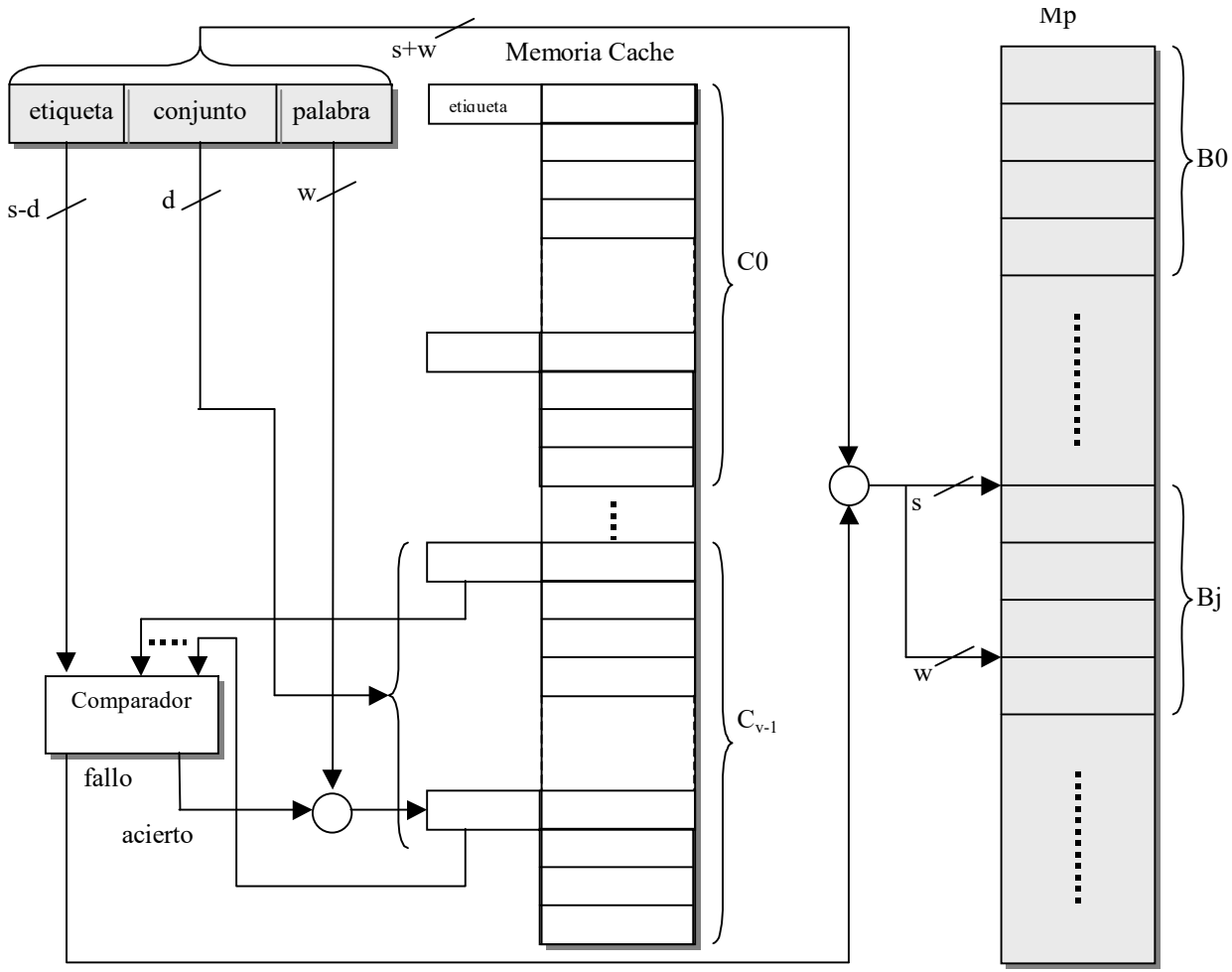
- La etiqueta tiene **s – d** bits para poder diferenciar a cada uno de los bloques de *Mp* que pueden ubicarse en el mismo conjunto de *Mc*.

- El directorio caché en correspondencia asociativa por conjuntos contendrá, pues, un registro de **s - d** bits por cada conjunto de líneas de *Mc*.

4. Elementos de diseño

➢ Correspondencia asociativa por conjuntos (2)

- El esquema de acceso a una Mc con correspondencia asociativa por conjuntos es el siguiente:



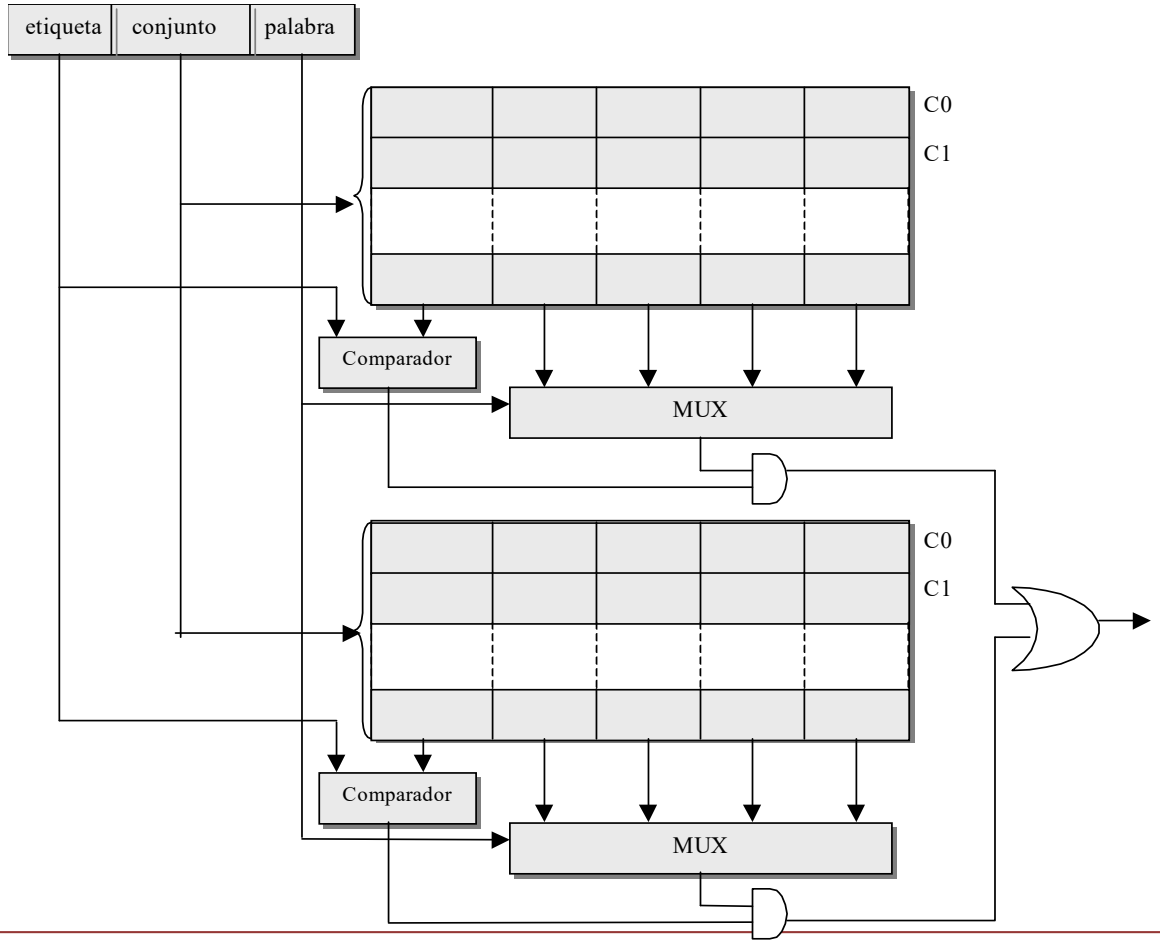
Los bits del campo *conjunto* de la dirección determinan el conjunto leído. Si la etiqueta de la dirección coincide con la etiqueta de alguna de las líneas del conjunto accedido, hay acierto, es decir, el bloque contenido en dicha línea es el mismo al que pertenece la dirección que accede al sistema de memoria. En caso contrario se produce un fallo de caché.

4. Elementos de diseño

➢ Correspondencia asociativa por conjuntos (3)

- Una memoria de correspondencia asociativa de grado k se organiza como un conjunto de k memorias de correspondencia directa, cada una con su comparador.
- Para el caso k= 2 (2 vías) el esquema sería el siguiente:

Con los bits de *conjunto* se accede a cada una de las palabras de las k RAM del sistema. La etiqueta de la dirección se compara en paralelo con los campos etiqueta de todas las palabras leídas. Si alguna coincide, hay acierto y con los bits del campo *palabra* seleccionan la palabra específica del bloque



4. Elementos de diseño

➢ Correspondencia asociativa por conjuntos (4)

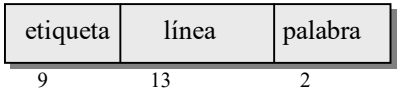
- **Ejemplo (anterior)**.

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

  Bloque = 4 bytes = 2^2 => w = 2

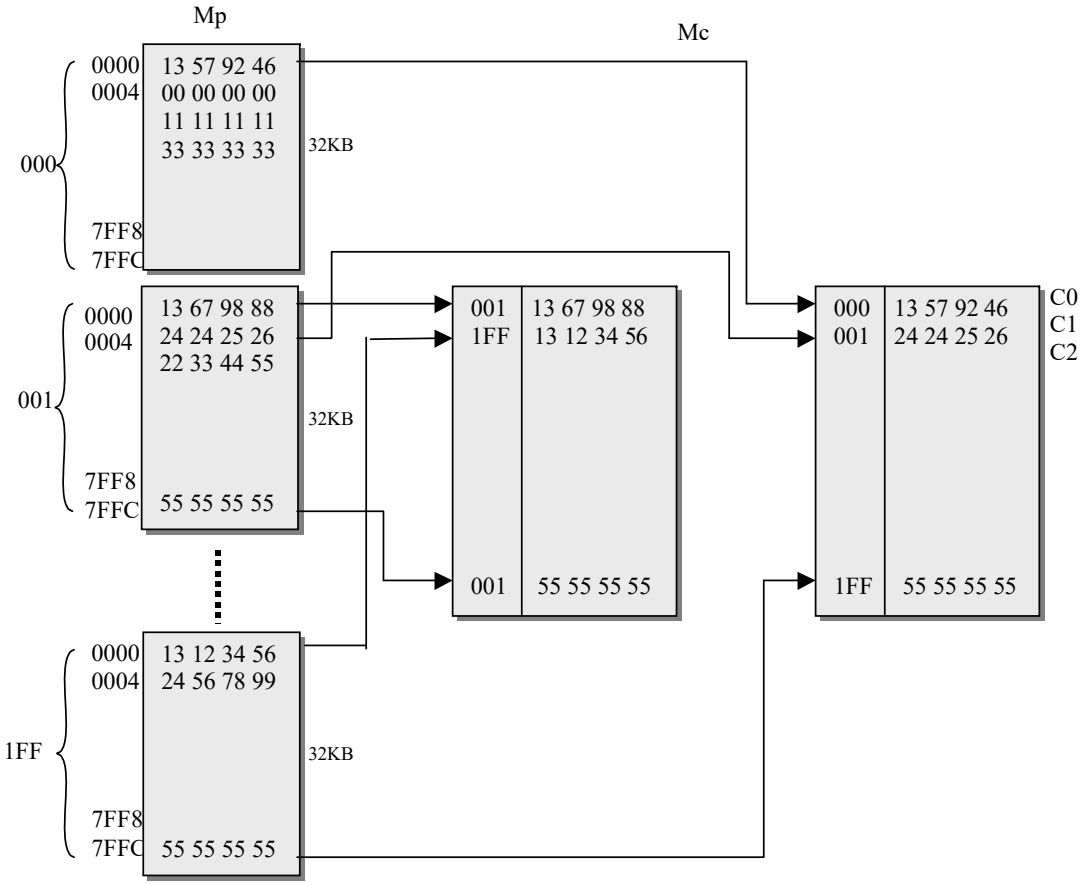
  Mc = 64 KBytes = 2^{16} Bytes = 2^{14} = 2^{14} líneas => r = 14

  Mp = 16 MBytes = 2^{24} Bytes = 2^{22} bloques => s = 22



El dibujo de la derecha muestra un posible estado de Mc y Mp con correspondencia asociativa por conjuntos de grado de asociatividad k = 2

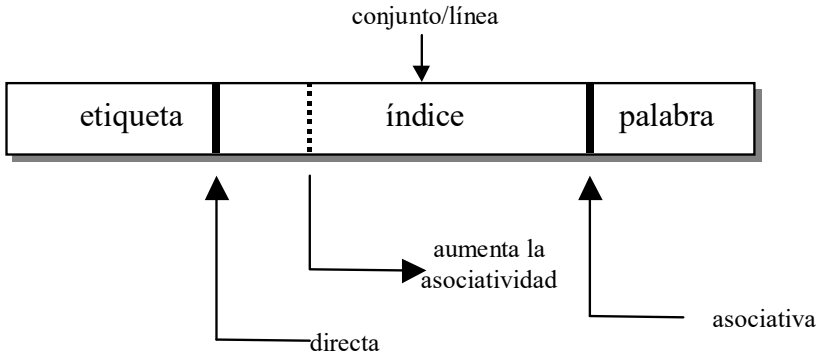
En este caso Mp se ha contemplado dividida en zonas con un número de blques igual al número de conjuntos de Mc, para facilitar gráficamente la correspondencia



4. Elementos de diseño

➢ Funciones de correspondencia unificadas: asociativa por conjunto

- 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:



Correspondencia	Ventajas	Inconvenientes
Directa	Acceso simple y rápido	Alta tasa de fallos cuando varios bloques compiten por el mismo marco
Asociativa	Máximo aprovechamiento de la MC	Una alta asociatividad impacta directamente en el acceso a la MC
Asociativa por conjuntos	Es un enfoque intermedio entre emplazamiento directo y asociativo El grado de asociatividad afecta al rendimiento, al aumentar el grado de asociatividad disminuyen los fallos por competencia por un marco Grado óptimo: entre 2 y 16 Grado más común: 2	Al aumentar el grado de asociatividad aumenta el tiempo de acceso y el coste hardware

4. Elementos de diseño

➢ Algoritmos de sustitución (o reemplazamiento)

- En las correspondencias asociativa y asociativa por conjuntos decide qué bloque sustituir.
- En la primera la elección se tiene que realizar sobre cualquier bloque presente en *Mc*
- En la segunda se reduce al conjunto único donde puede ubicarse el nuevo bloque.
- Las políticas de reemplazamiento más utilizadas son cuatro:

  • **Aleatoria**: se escoge un bloque al azar

  • **LRU(*Least Recently Used*)**:Se sustituye el bloque que hace más tiempo que no ha sido referenciado
  Algoritmo: se asocian contadores a las líneas y Si se referencia *MBi*

  MBk : contador(MBk) ≤ contador(MBi) ==> contador(MBk) := contado(MBk) + 1
  contador(MBi) = 0

  Cuando se produce un fallo se sustituye el MBi : contador(MBi) = MAXIMO

  Memoria asociativa de dos vías: basta con añadir un **bit de uso** a cada marco de bloque.

  • Cuando se referencia una línea su bit de uso se pone a 1 y el de la línea del mismo conjunto a 0.

  • Cuando hay que sustituir se elige el marco de bloque con bit de uso igual a 0.

  • ***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

  • ***LFU(Least Frequently Used):*** Se sustituye aquel bloque que ha experimentado menos referencias
  Se puede implementar asociando un contador a cada marco de bloque

4. Elementos de diseño

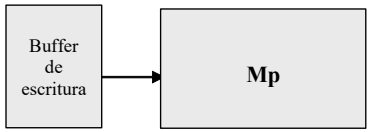
➢ Política de escritura

- Determina la forma de actualizar Mp cuando se realizan operaciones de escritura.
- 2 casos: cuando la posición en la que se escribe está en Mc (acierto) y cuando no está (fallo).

• **Frente a aciertos en la caché**: dos alternativas:

✓ **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)



✓**Postescritura** (***copy back***)

- Las actualizaciones se hacen sólo en *Mc*
- Se utiliza un bit de actualización asociado a cada marco de bloque
- Inconveniente: inconsistencia temporal entre Mc y Mp

• **Frente a fallos en la caché**

✓ **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*

✓**No asignación en escritura** (***No write allocate***)**:** el bloque se modifica en Mp sin cargarse en Mc

4. Elementos de diseño

➢ Política de búsqueda

- Determina las condiciones que tienen que darse para buscar un bloque de Mp y llevarlo a Mc.
- Existen dos alternativas principales:

- **Por demanda**: se lleva un bloque a Mc cuando se referencia alguna palabra del bloque y éste no está en Mc

- **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 se buscan los bloques Bi y Bi+1

- **Clasificación de los fallos caché: l**os 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**: producidos por la necesidad de ubicar un bloque en un conjunto lleno cuando Mc no está llena

5. Mejora del rendimiento de la memoria caché

➢ Factores que determinan el rendimiento de la memoria caché

- **Tiempo de acceso** a memoria durante la ejecución de un programa:

  Tacceso = Na * Tc + Nf * Tp

  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*

- Tiempo de acceso medio durante la ejecución del programa valdrá:

  Tacceso_medio = Tacceso/ Nr = Ta*Tc + Tf*Tp

  Ta = Na/Nr (tasa de aciertos)
  Tf = Nf/Nr (tasa de fallos)
  Nr = Na+Nf = número total de referencias a memoria

  Tacceso_medio = Tacierto + Tf * Tp

*Tacierto = Ta*Tc*, si frente a un fallo, el bloque se lleva a Mc al tiempo que a la CPU

Tacierto = Tc, si frente a un fallo, el bloque se lleva a Mc y después (secuencia) a la CPU

*(Tacceso = Nr * Tc + Nf * Tp ==>* ***Tacceso_medio = Tacceso/ Nr = Tc + Tf * Tp)***

5. Mejora del rendimiento de la memoria caché

➢ Alternativas para mejorar el rendimiento de una caché

Tacceso_medio = Tacierto + Tf * Tp

- Reducción de la tasa de fallos: ***Tf***
  - Ajuste del tamaño de bloque
  - Ajuste de la asociatividad
  - Caché seudoasociativa
  - Caché víctima
  - Pre-búsqueda hardware
  - Pre-búsqueda software
  - Optimización del código

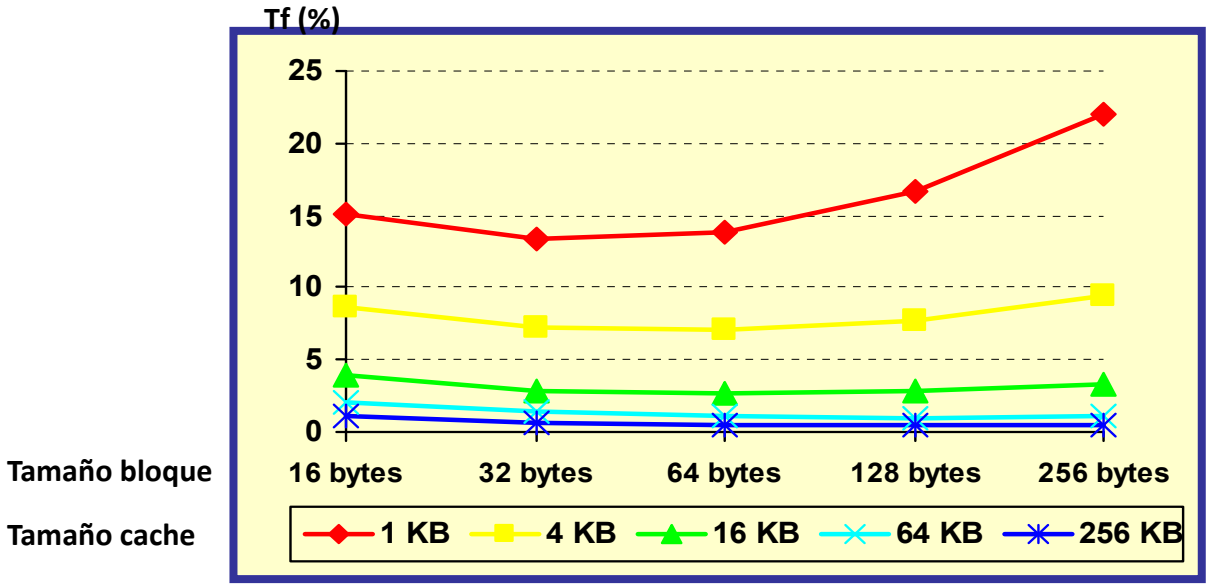
- Reducción de la penalización de los fallos: ***Tp***
  - *Cachés multinivel*
  - *Prioridad de las lecturas frente a las escrituras*

- Reducción del tiempo de acierto: ***Tacierto***
  - *Caché pequeña y sencilla*

5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: ajuste del tamaño de bloque

- Aumentar el tamaño del bloque disminuye la tasa de fallos iniciales y **captura mejor la localidad espacial**
- Pero se aumenta la tasa de fallos de capacidad (menor Nº Bloques => **captura peor localidad temporal**)
- Al tener bloques más grandes aumenta la penalización por fallos



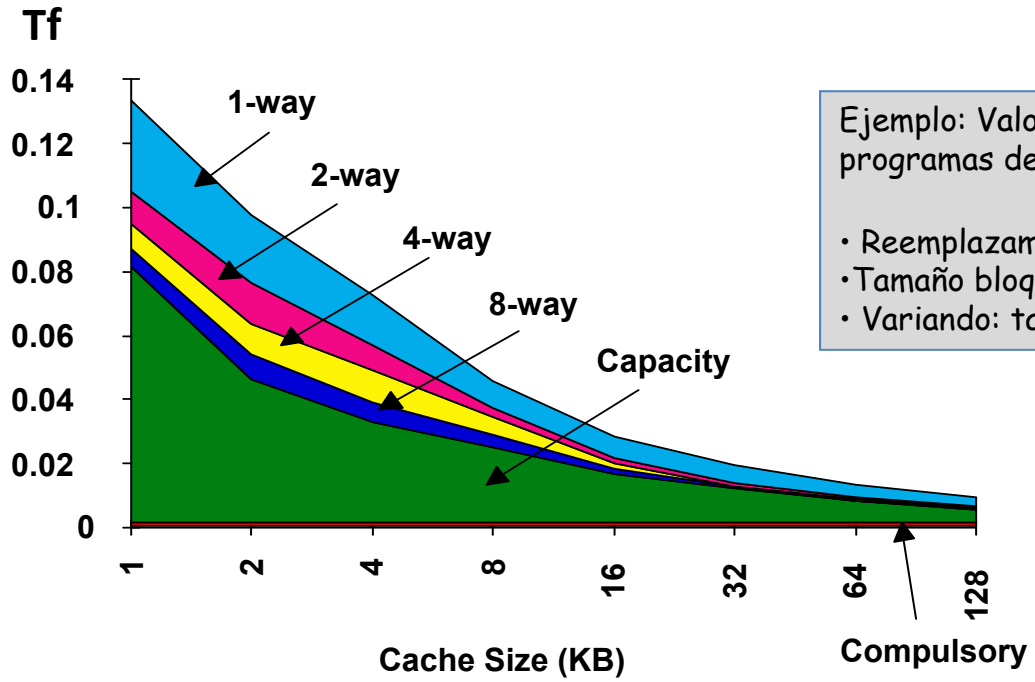
5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: ajuste 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*



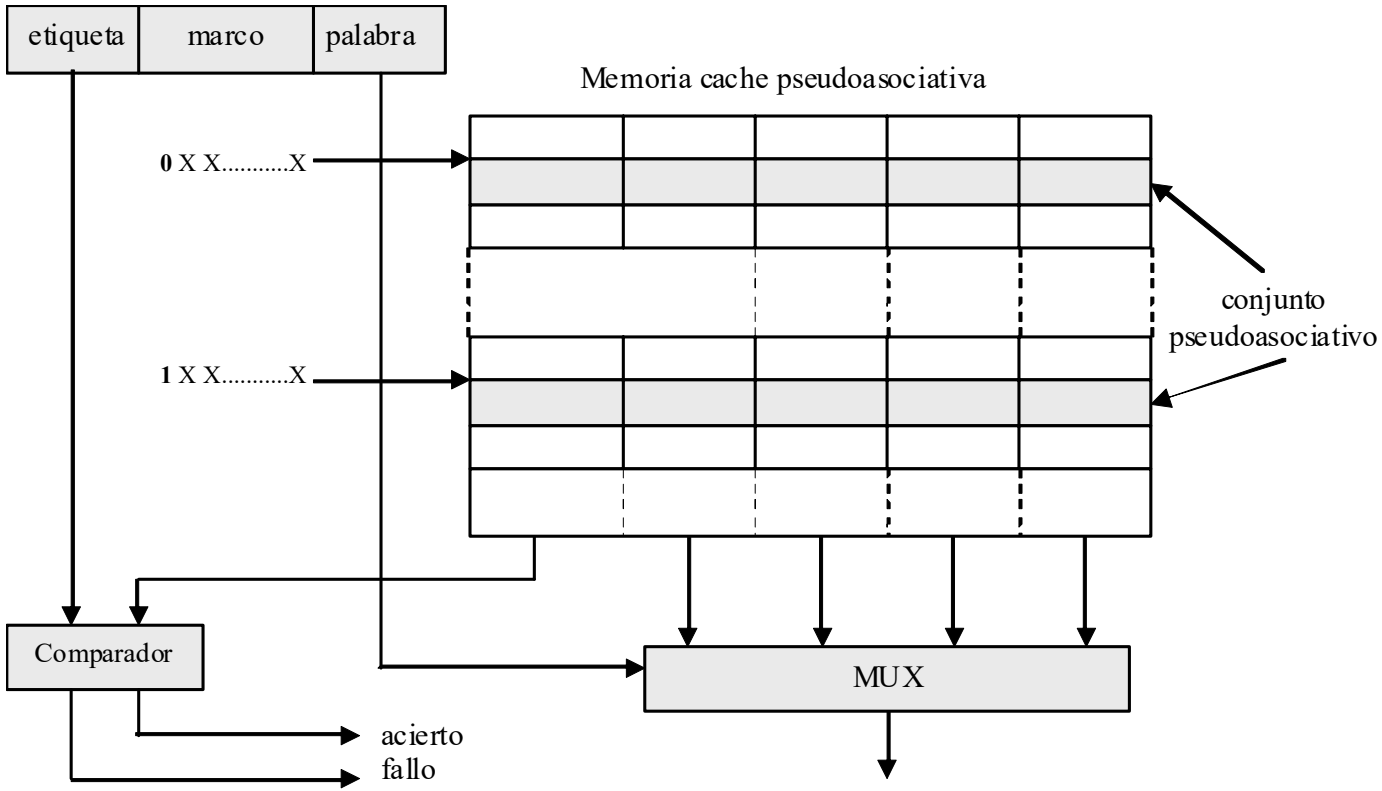
Ejemplo: Valores promedio de Miss Rate para programas de SPEC92

- Reemplazamiento: LRU
- Tamaño bloque: 32 bytes (cte)
- Variando: tamaño cache, asociatividad

5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: caché seudoasociativa

- Caché de correspondencia directa con una modificación para que se comporte como asociativa
- Se permite que un bloque de Mp se pueda ubicar en dos (pseudoasociativa de 2 vías) marcos de Mc:
  - el que le corresponde (por la correspondencia directa)
  - el que resulta de conmutar el bit más significativo de la dirección del bloque



5. Mejora del rendimiento de la memoria caché

➢ Rendimiento de una caché pseudoasociativa

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

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

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

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

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

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

$Tiempo\_acceso\_medio_{pseudo} =$

$Tiempo\_acierto_{directo} + (Tasa\_fallos_{directa} - Tasa\_fallos_{2vias}) * 2 + Tasa\_fallos_{2vías} * Penalización\_fallos_{directa}$

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	50	1

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

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

5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Cache Víctima

- Se propuso por primera vez en 1990, la idea es utilizar una cache pequeña completamente asociativa (CV) conjuntamente con una cache grande con una asociatividad más limitada (MC)

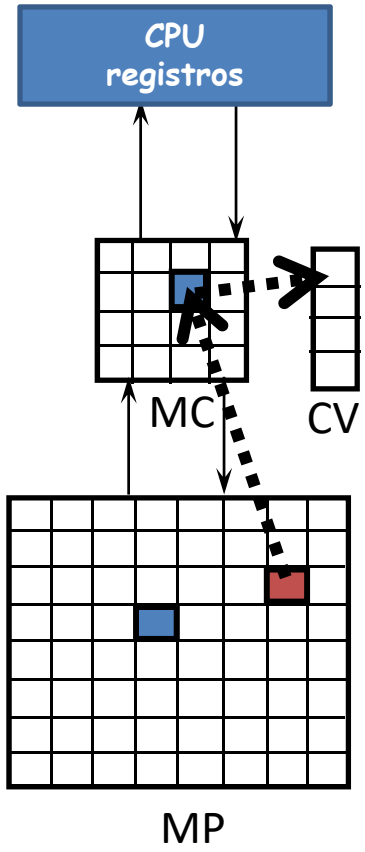
- Se intenta conseguir que MC tenga la misma tasa de fallos que si fuera completamente asociativa

  - Ej: MC de acceso directo + CV asociativa tiene la misma tasa de fallos que MC con 2 vías. Un bloque puede estar en MC o en CV pero nunca en las dos

- En la cache víctima estarán aquellos bloques que han sido re-emplazados de la memoria cache

  - Un bloque pasa de MC a CV cuando se produce un re-emplazo

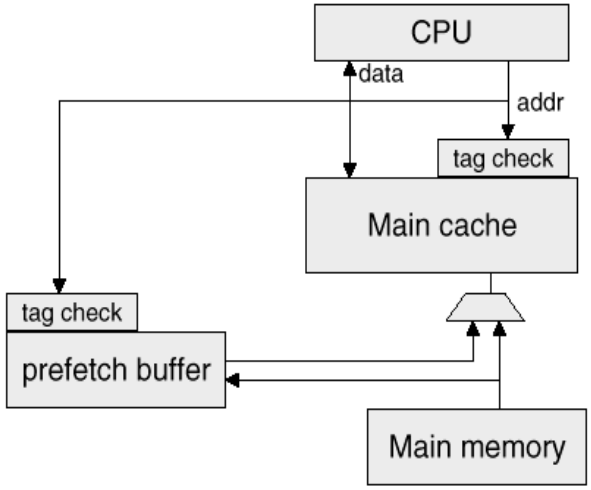
  - Un bloque pasa de CV a MC cuando es referenciado



5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Pre-búsqueda Hardware

- Anticipa los fallos de Cache anticipando las búsquedas antes de que el procesador demande el dato o la instrucción que provocarían un fallo
- Típicamente la CPU busca dos bloques en un fallo (el referenciado y el siguiente)
- El bloque buscado se lleva a Mc
- El prebuscado se lleva a un buffer ("prefetch buffer" o "stream buffer"). Al ser referenciado pasa a MC



5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Pre-búsqueda Software

- Se introducen en el hardware 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.

- La eficiencia depende del compilador y del tipo de programa

- Pre-búsqueda con destino en cache (MIPS IV, PowerPC, SPARC v. 9), o en registro (HP-PA)

- Funciona bien con bucles y patrones simples de acceso a arrays. Aplicaciones de cálculo

- Funciona mal con aplicaciones enteras que presentan una amplia reutilización de Cache

- Recargo (overhead) por las nuevas instrucciones. Más búsquedas. Más ocupación de memoria

5. Mejora del rendimiento de la memoria caché

➢ Pre-búsqueda Software: ejemplo

- 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];
```

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 elemento de los *arrays a[i][j]* y *b[i][j]* ocupan 8 bytes
- Están dispuestos en memoria en orden ascendente de sus índices:

5. Mejora del rendimiento de la memoria caché

➢ Pre-búsqueda Software: ejemplo (sin prebúsqueda)

▪ **Acceso a *a[ ][ ]***

- Se beneficia de la localidad espacial
- Valores pares de *j* ➔ fallos (forzosos)
- Valores impares de *j* ➔ aciertos (van a Mc en los bloques de los pares).

Número de fallos = *(3 * 100)/2 = 150.*
(3 filas y 100 columnas)

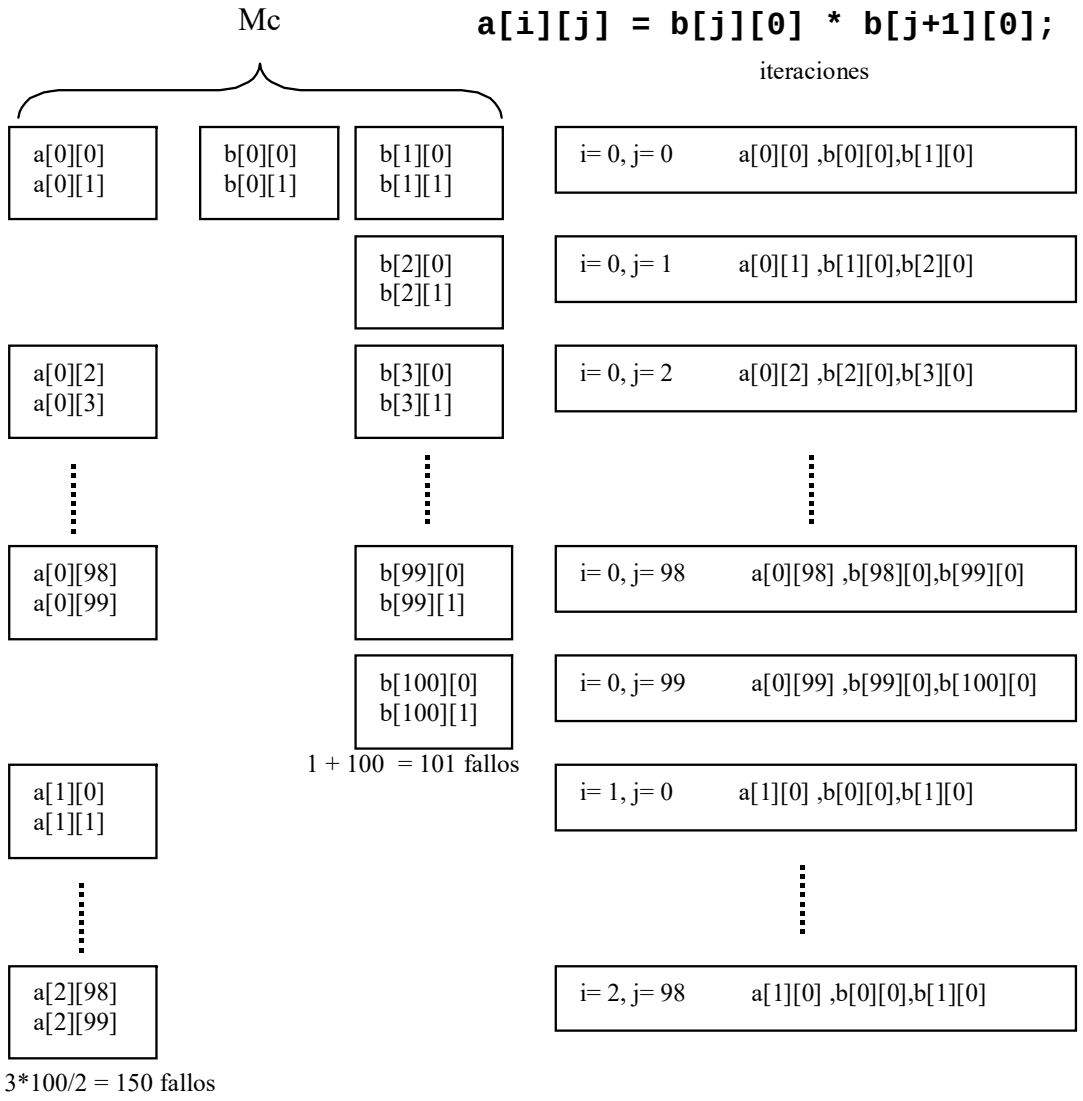
▪ **Acceso a *b[ ][ ]***

- No se beneficia de la localidad espacial: Los accesos no se realizan en el orden en que están almacenados.
- Se beneficia dos veces de la temporal: Se accede a los mismos elementos para cada iteración de *i*
- Cada iteración de *j* usa los mismos valores de *b[ ][ ]* en la anterior.

Número de fallos = 101

1 para *b[0][0]*
100 para *b[1][0]* hasta *b[100][0]).*

Número total de fallos = 251.



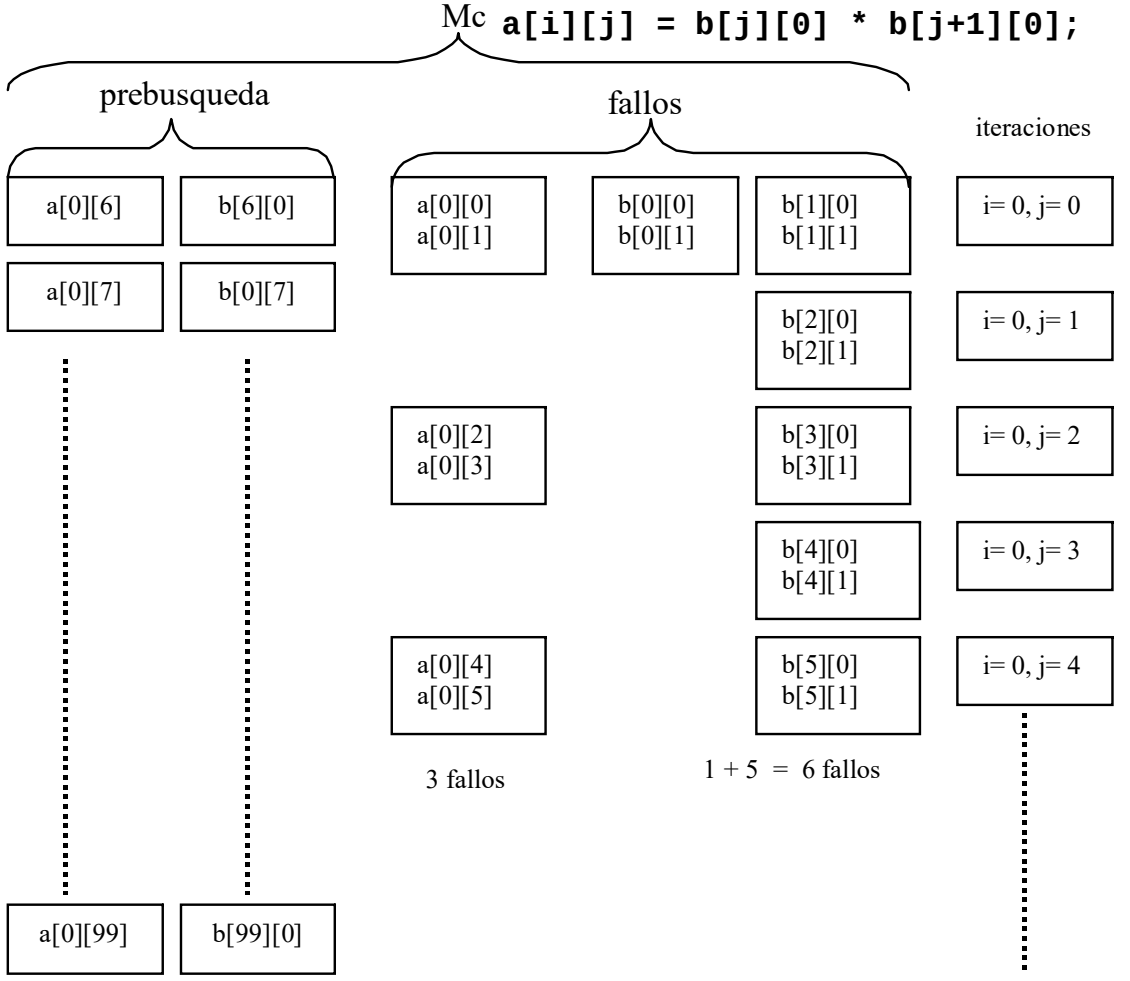
5. Mejora del rendimiento de la memoria caché

➢ Pre-búsqueda Software: ejemplo (con prebúsqueda)

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

```
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 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] b[i][j] ya se ha prebuscado en el primero.
- Fallos en las 3 iteraciones de *a[][]* = 3*3=9
- Fallos en *b[][]* =6
- **Número total de fallos = 6 + 9 = 15**

5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Optimización del código (1)

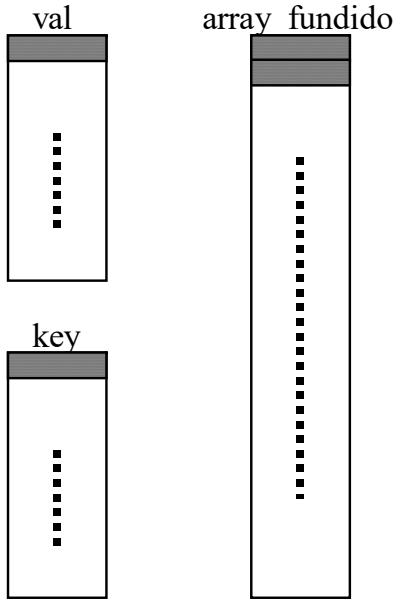
▪ **Fusión de *arrays***

• 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.

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 *arrays* originales.

5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Optimización del código (2)

- **Fusión de bucles**

Ejemplo:

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 el programa transformado estas referencias se hacen para los mismos valores de los índices en las 2 expresiones consecutivas.

5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Optimización del código (3)

- **Intercambio de bucles**

Ejemplo:

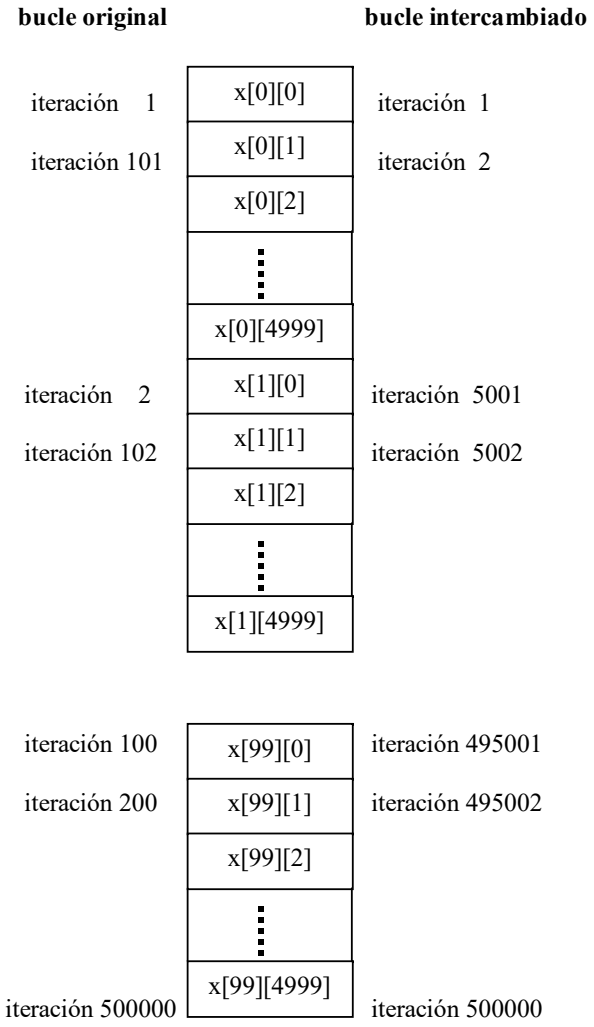
programa original

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

programa transformado

```
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.



5. Mejora del rendimiento de la memoria caché

➢ Reducción de la tasa de fallos: Optimización del código (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* opera 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 = 0; 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;
};
```

5. Mejora del rendimiento de la memoria caché

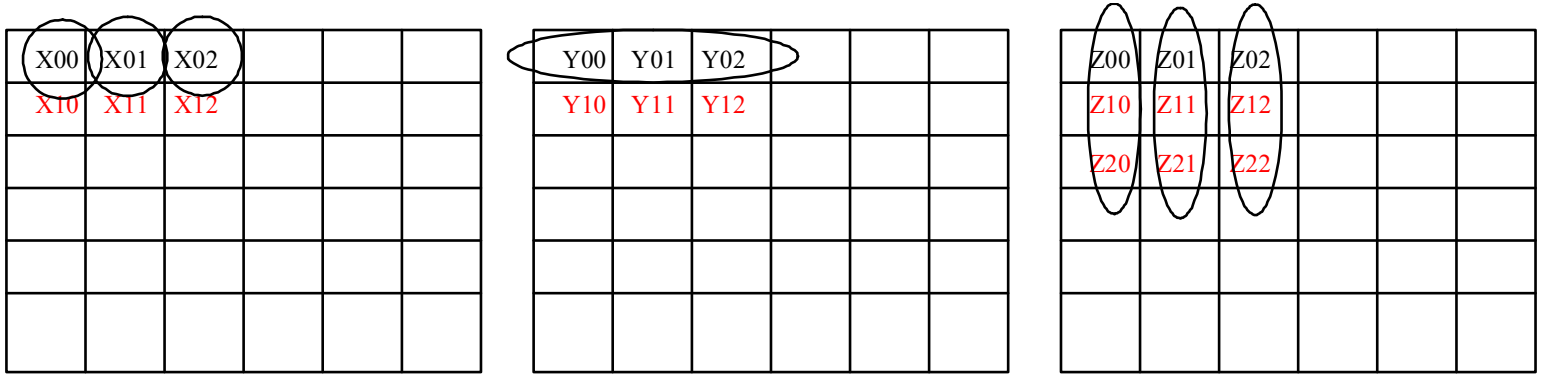
➢ Reducción de la tasa de fallos: Optimización del código (5)

- **Ejemplo:** multiplicación de matrices

  • **Programa transformado**

```
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;
};
```

• Calcula 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:



5. Mejora del rendimiento de la memoria caché

➢ Reducción de la penalización por fallo: Cachés Multinivel

▪ 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.

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

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

▪ Con varios niveles de cachés hay que diferenciar la **tasa de fallos local de la global**:

Tasa_fallos_local = nº de fallos / nº de accesos a caché

Tasa_fallos_global = nº de fallos / nº total de accesos desde la CPU

- En general se cumple:

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

- *Y en particular:*

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

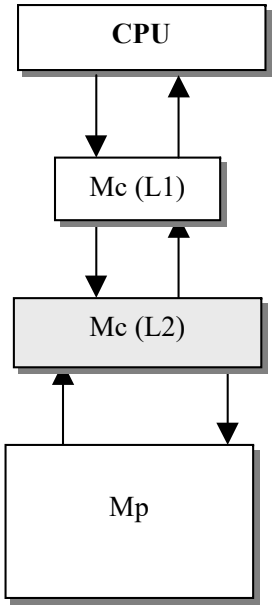
$$Tasa\_fallos\_local_{N2} > Tasa\_fallos\_global_{N2}$$

Ejemplo:
1000 referencias (cahé de dos niveles)
40 fallos se producen en N1
20 fallos se producen en N2

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

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

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



5. Mejora del rendimiento de la memoria caché

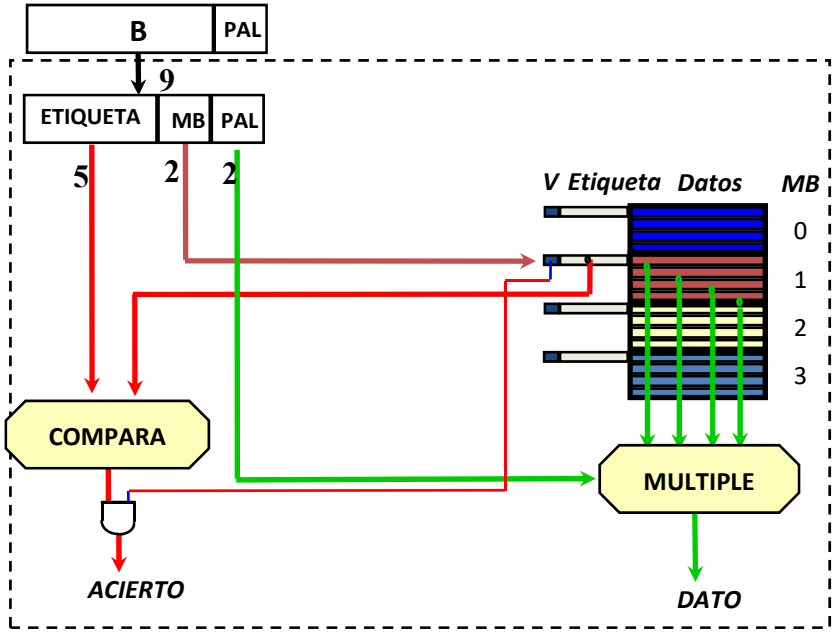
➢ Reducción de la penalización por fallo: prioridad de lecturas sobre escrituras

- Un fallo de lectura puede impedir la continuación de la ejecución del programa; un fallo de escritura puede ocultarse.
- Buffer de escrituras (rápido). Depositar en buffer las palabras que tienen que ser actualizadas en MP y continuar ejecución.
- La transferencia del buffer a MP se realiza en paralelo con la ejecución del programa.
- PROBLEMA: podemos estar intentando leer una palabra que todavía está en el buffer.

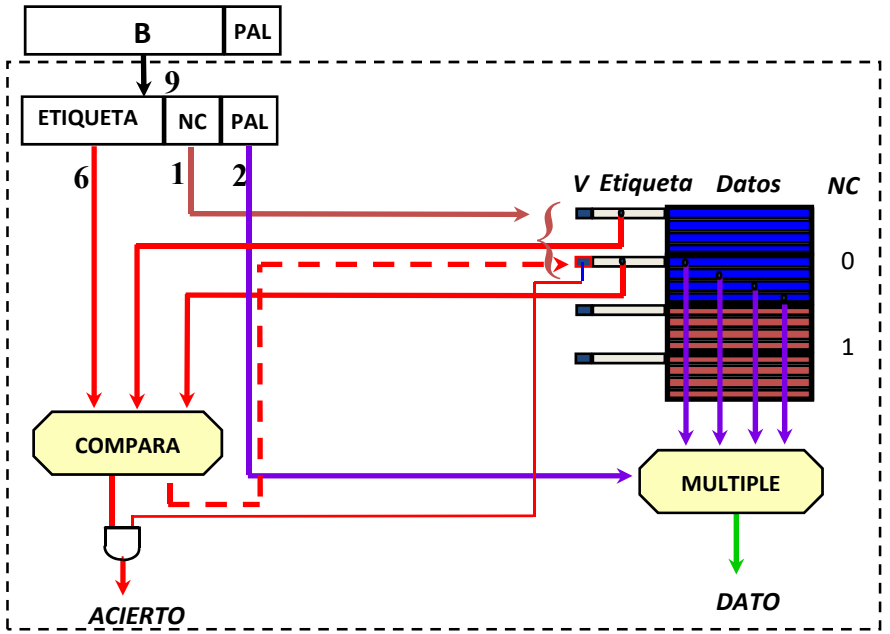
5. Mejora del rendimiento de la memoria caché

➢ Reducción del tiempo de acierto: Cache pequeña y sencilla (1)

- El acceso al directorio y la comparación de etiquetas consume tiempo
- Ejemplo: Comparación de acceso a un dato en cache directa y en cache asociativa por conjuntos con 2 vías



Directo



Asociativo por conjuntos 2 vías

5. Mejora del rendimiento de la memoria caché

➢ Reducción del tiempo de acierto: Cache pequeña y sencilla (2)

- Una cache pequeña se pueda integrar junto al procesador
  - evitando la penalización en tiempo del acceso al exterior
  - Tiempo de propagación versus tiempo de ciclo del procesador
- Ejemplo: tres generaciones del procesadores AMD (K6, Athlon y Opteron) han mantenido el mismo tamaño para las caches L1
- Simple (cache directa o grado de asociatividad pequeño)
  - En cache directa se puede solapar chequeo de tags y acceso al dato, puesto que el dato sólo puede estar en un lugar
  - El aumento del número de vías puede aumentar los tiempos de comparación de tags
- Ejemplo: impacto del tamaño de la cache y la asociatividad sobre el tiempo de acceso (tecnología 90 nm)