

# Tema 4. Sistema de Memoria de Altas Prestaciones

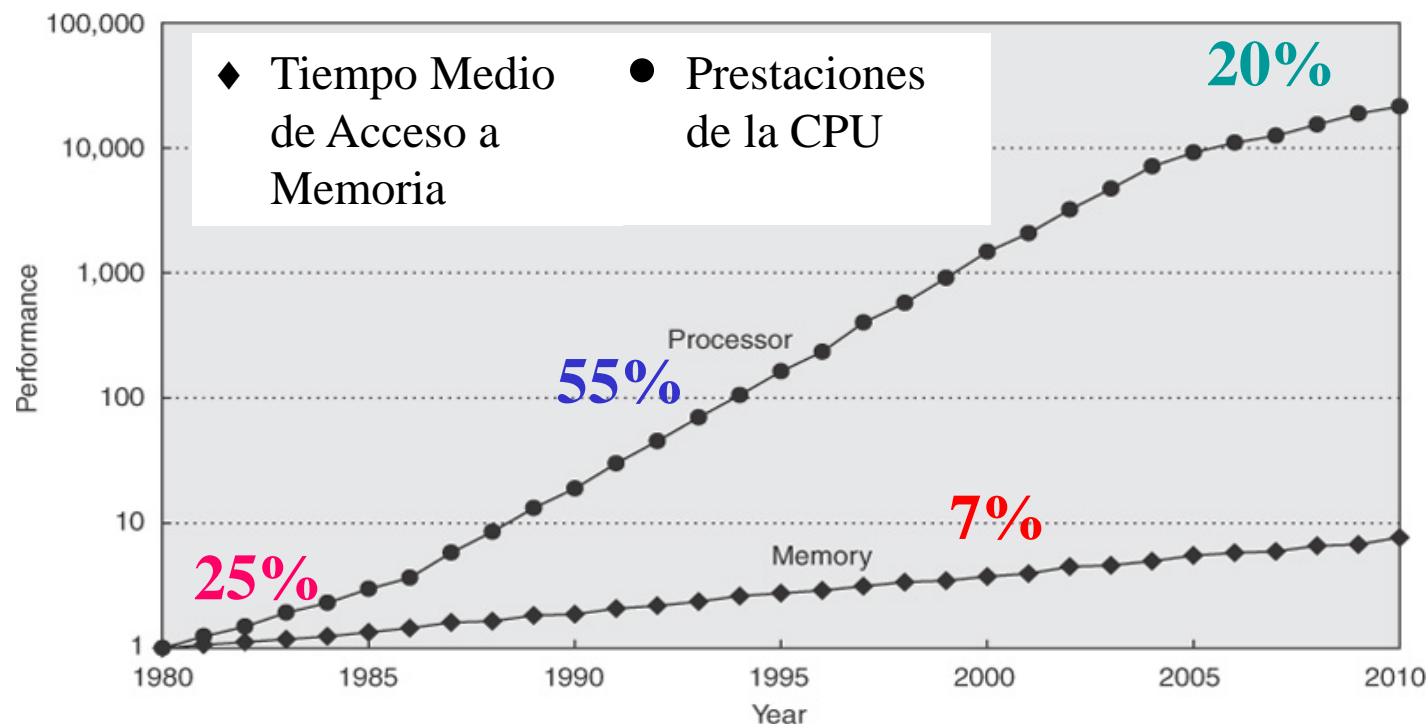
Departamento de Ingeniería y  
Tecnología de Computadores

# Índice

- Introducción
- Evaluación del Rendimiento de la Jerarquía de Memoria
- Reducción de la Tasa de Fallos de Caché
- Reducción de la Penalización por Fallo de Caché
- Reducción del Tiempo en Caso de Acierto en Caché
- Organizaciones de la Memoria Principal

# Introducción

- La gran diferencia en el crecimiento de las prestaciones entre los procesadores y la memoria ha obligado a estudiar nuevas medidas para mejorar los subsistemas de memoria

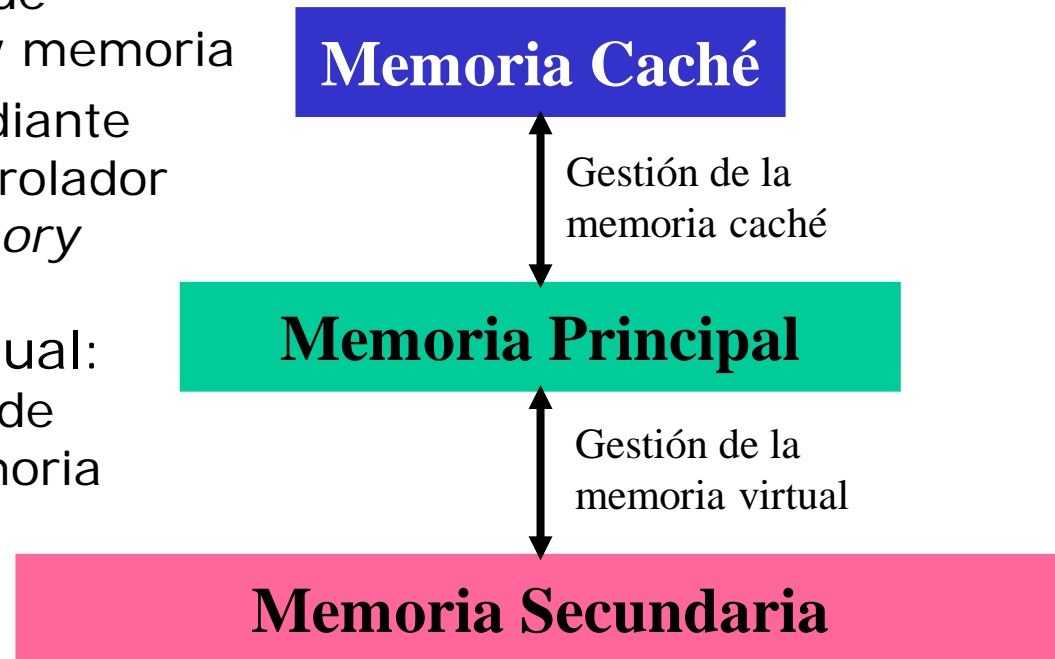


# Introducción

- Un computador típico contiene una jerarquía de memoria formada por:
  - Registros de la CPU
  - Memoria caché (varios niveles)
  - Memoria principal
  - Memoria secundaria (discos)
  - Memoria de almacenamiento masivo (cintas, DVD, ...)
- El coste de todo el sistema de memoria excede al de la CPU, así pues es importante optimizar su rendimiento
- Objetivos de la jerarquía de memoria:
  - Hacer que el usuario tenga la ilusión de que dispone de una memoria con:
    - Tiempo de acceso similar al nivel más rápido.
    - Coste por bit similar al del nivel más barato.

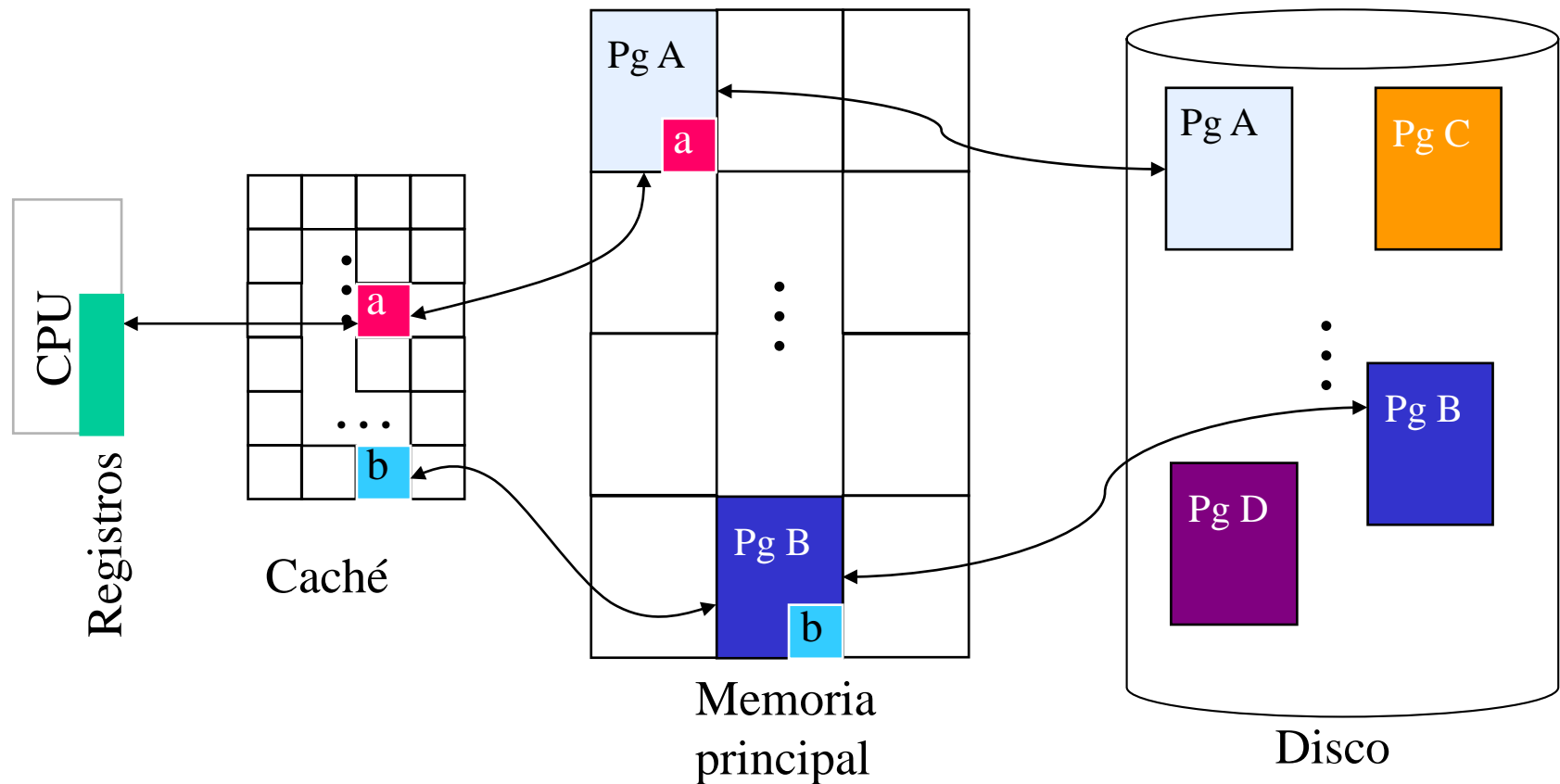
# Introducción

- La gestión en tiempo de ejecución de la jerarquía de memoria afecta a los niveles de **memoria caché**, **memoria principal** y **memoria secundaria**
  - Los registros del procesador normalmente los asigna el compilador y la memoria de almacenamiento masivo se usan para backup
- Gestión de la memoria caché:
  - Controla la transferencia de información entre caché y memoria
  - Suele llevarse a cabo mediante hardware específico (controlador de caché y la MMU - *Memory Management Unit*)
- Gestión de la memoria virtual:
  - Controla la transferencia de información entre la memoria secundaria y la principal
  - Realizada mediante una combinación hardware (MMU) software (SO)



# Introducción

- La memoria caché retiene información recientemente usada y también cercana a la recientemente usada
- Tanto la memoria principal como la memoria caché se dividen en bloques de igual tamaño (bloques vs páginas)



# Introducción

- **Inclusión:**

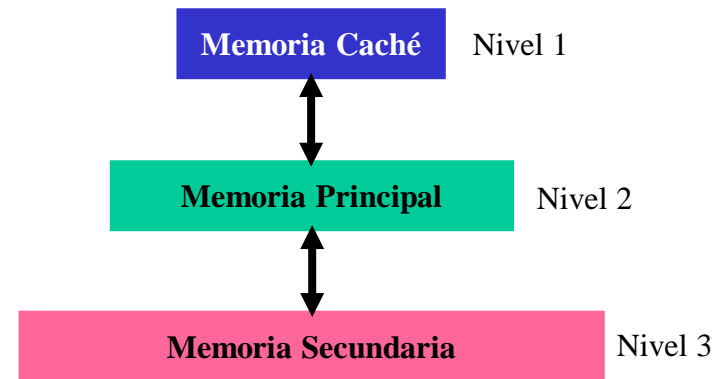
- Cualquier información almacenada en un nivel de memoria, debe encontrarse también en los niveles inferiores

- **Coherencia (entre niveles):**

- Si un bloque de información se actualiza en un nivel, deben actualizarse los niveles inferiores:
  - **Post-escritura** (*write-back*): la actualización del siguiente nivel se retrasa hasta que el bloque que se modificó se expulsa
  - **Escritura directa** (*write-through*): Cuando una palabra se modifica en un nivel, inmediatamente se actualiza el siguiente

- **Localidad:** Las referencias a memoria (datos e instrucciones) se concentran en regiones del tiempo y del espacio

- **Localidad temporal:** Las posiciones de memoria recientemente referenciadas, serán referenciadas de nuevo en un futuro próximo
- **Localidad espacial:** Tendencia a referenciar elementos de memoria cercanos a los últimos elementos referenciados



# Introducción

- **Terminología:**

- **Bloque:** unidad mínima de transferencia entre dos niveles
- **Acierto** (hit): el dato solicitado está en el nivel  $i$ 
  - Tasa de aciertos (hit ratio): fracción de accesos encontrados en el nivel  $i$
  - Tiempo de servicio en caso de acierto: tiempo del acceso al nivel  $i$  más el tiempo de detección del acierto
- **Fallo** (miss): el dato solicitado no está en el nivel  $i$  y es necesario buscarlo en el nivel  $i+1$ 
  - Tasa de fallos (miss ratio):  $1 - \text{Tasa de aciertos}$
  - Tiempo de penalización por fallo: tiempo de sustitución de un bloque del nivel  $i$  más el tiempo de acceso al dato
- El tiempo de servicio en caso de acierto debe ser mucho menor que el tiempo de penalización en caso de fallo



# Repaso - Cachés

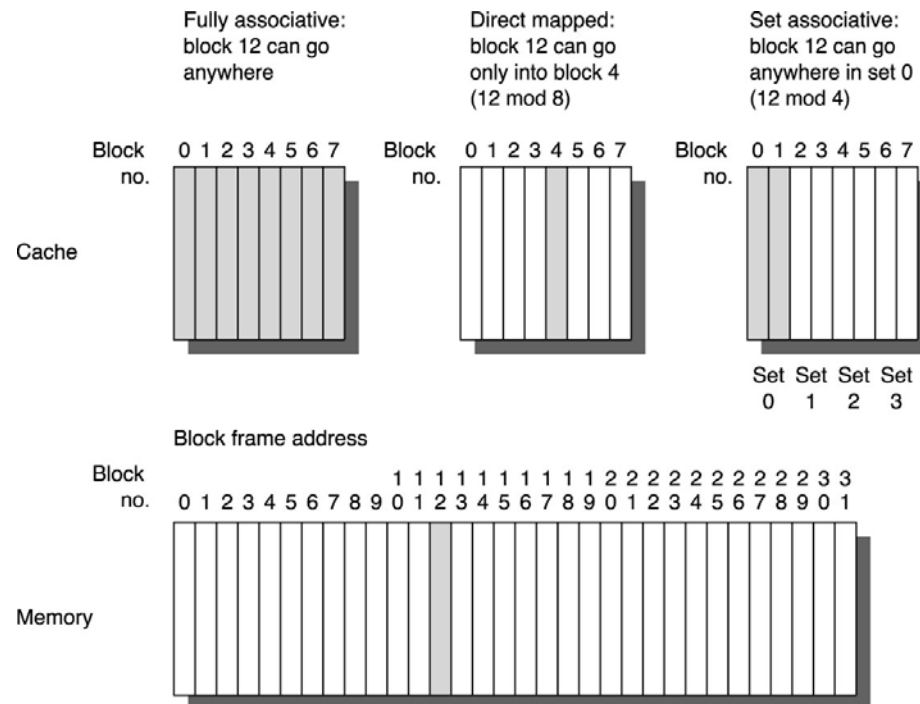
- **REPASO:**

*las cuatro cuestiones de una caché*

1. ¿Dónde puede situarse un bloque en una caché?
2. ¿Cómo sabemos si un bloque está en la caché?
3. ¿Qué bloque debe ser reemplazado en caso de fallo de caché?
4. ¿Qué ocurre en las escrituras?

# 1. ¿Dónde puede situarse un bloque en una caché?

- **Correspondencia directa:** conjuntos de un solo bloque  $\Rightarrow$  cada dato sólo puede colocarse en una posición de la caché
- **Totalmente asociativa:** en cualquier bloque de la caché
- **Asociativa por conjuntos:** cada conjunto comprende varios bloques de la caché. El bloque va a un conjunto y luego se puede situar en cualquier hueco del conjunto. Si hay  $n$  bloques en un conjunto se llama **asociativa por conjuntos de  $n$  vías**



## 2. ¿Cómo sabemos si un bloque está en la caché?

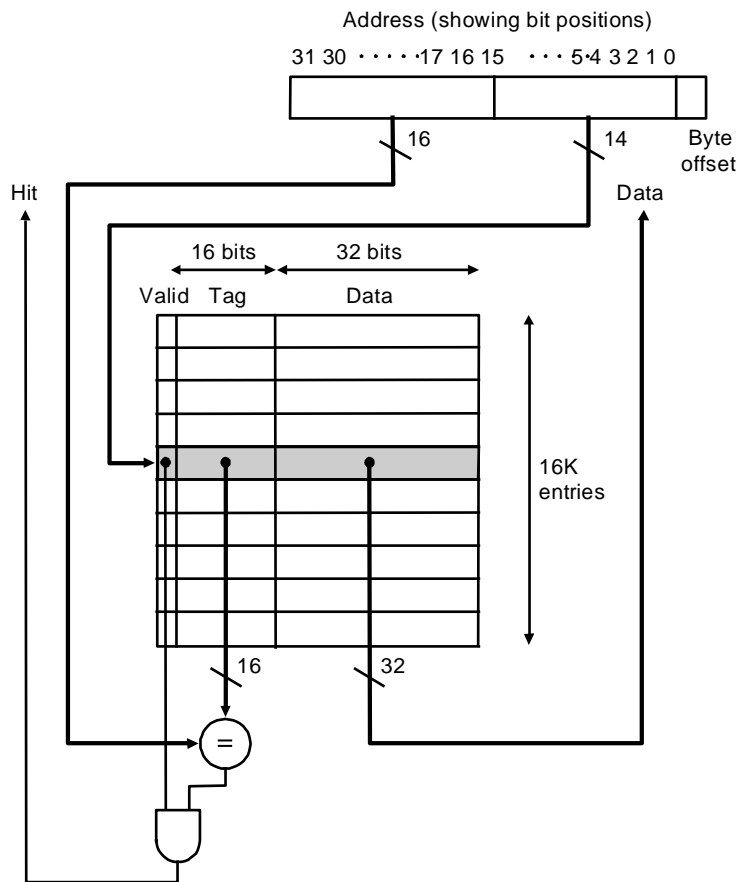
| Block address |       | Block offset |
|---------------|-------|--------------|
| Tag           | Index |              |

- De la dirección del dato de memoria buscado se extraen los campos índice, etiqueta y desplazamiento (offset)
  - El campo índice selecciona el conjunto donde puede encontrarse el bloque
  - La etiqueta diferencia todos los bloques de memoria que mapean al mismo conjunto
  - El offset selecciona la palabra o byte deseado dentro del bloque
  - Si el tamaño de la caché no cambia, aumentar la asociatividad, hace que disminuya el tamaño del campo índice y aumente el de la etiqueta
- Cada bloque de la caché tiene un bit de validez que indica si el bloque contiene información válida
- Además, tiene asociada una etiqueta para identificar cuál es el bloque de memoria actualmente almacenado
- Cuando se busca un bloque en la caché, las etiquetas de todos los bloques del conjunto se comparan en paralelo con la etiqueta de la dirección buscada

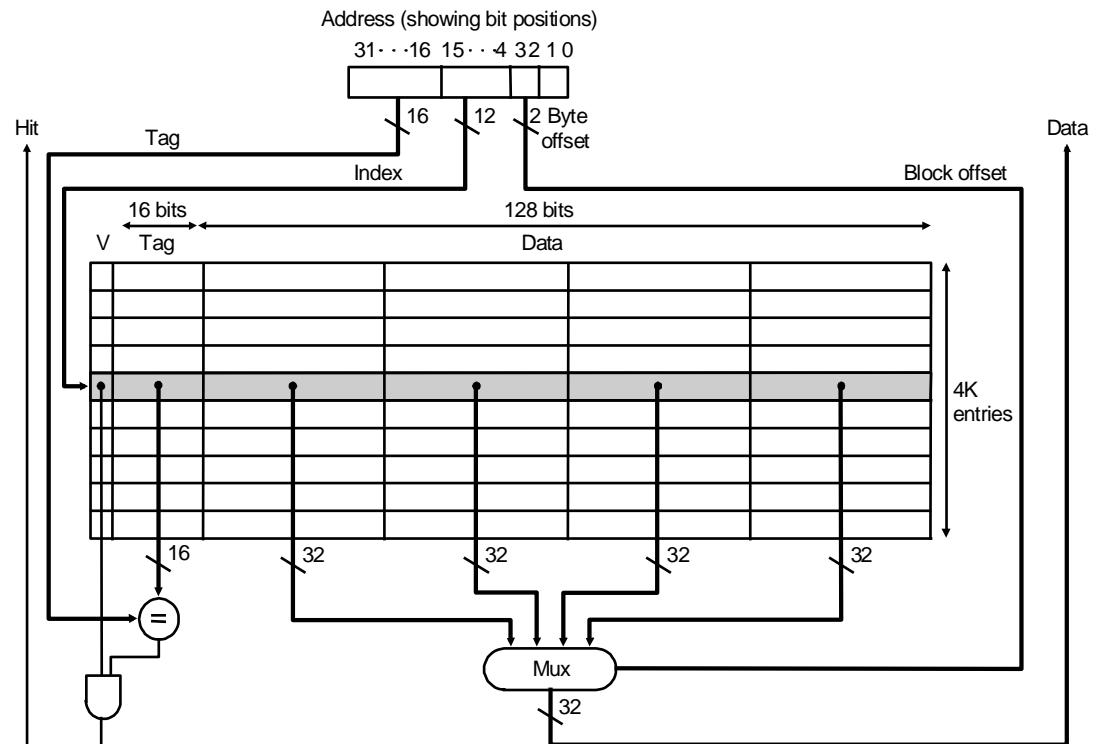
# Ejemplos de correspondencia directa

- Caché de 16 K palabras de 32 bits:

## Bloque de 1 palabra



## Bloque de 4 palabras



### 3. ¿Qué bloque se reemplaza en caso de fallo?

- Cuando ocurre un fallo, el controlador de memoria debe seleccionar un bloque para ser sustituido
  - **Correspondencia directa**
    - sólo hay un bloque posible por lo que no se aplica ninguna política de reemplazo
  - **Asociativas ( $>1$ ):** varios bloques candidatos a ser reemplazados. Se aplica una **política de reemplazo**:
    - **Aleatoria**: sencilla de implementar
    - **LRU**: bloque menos recientemente usado. Más efectiva, pero al aumentar el número de bloques por conjunto se vuelve más costosa
    - **FIFO**: se elige por orden de antigüedad. El primero que entró es el candidato a ser expulsado

## 4. ¿Qué ocurre en las escrituras?

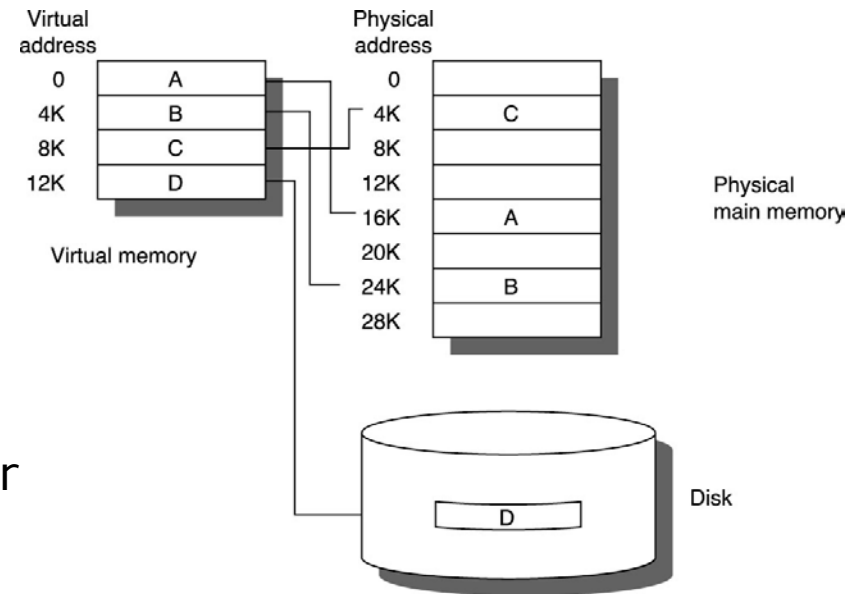
- Las escrituras son **más difíciles** de implementar:
  - No se puede comenzar a modificar un bloque hasta que no se ha comparado la etiqueta
  - Normalmente se modifica **sólo una parte** del bloque
- **Políticas de escritura:**
  - **Escritura directa** (*write-through*): La información se escribe tanto en caché como en el siguiente nivel de memoria
    - Más fácil de implementar
    - El siguiente nivel está siempre actualizado
    - Sólo se escribe la palabra modificada, no el bloque completo
    - Puede provocar más esperas hasta que se realiza la escritura
    - Una optimización común es un **buffer de escritura** que permite al procesador continuar tan pronto los datos están en el buffer
  - **Post-escritura** (*write-back*): La información se escribe **solamente** en caché. Cuando el bloque es sustituido, se actualiza el siguiente nivel
    - Necesita un **bit de sucio**
    - Las escrituras se realizan a la velocidad de la caché y varias escrituras en un bloque sólo requieren una escritura en memoria
    - Necesita **menos ancho de banda**, ya que hay escrituras que no van a memoria

## 4. ¿Qué ocurre en las escrituras? (cont.)

- En caso de fallo de caché al escribir:
  - Tenemos dos políticas
    - **Carga en escritura** (*write-allocate*): Si cuando se va a escribir el dato no se encuentra el bloque en caché, se lee el bloque y se carga en caché
    - **No carga en escritura** (*no-write-allocate*): El bloque se modifica directamente en el nivel inferior y no se lleva a la caché
  - Lo normal es hacer las siguientes combinaciones:
    - Post-escritura y carga en escritura
    - Escritura directa y no carga en escritura

# Repaso - Memoria virtual

- La **memoria virtual** surge para permitir que los procesos manejen de forma automática un conjunto de datos mayor que la memoria real del sistema
- Permite que:
  - Cada proceso tenga su propio espacio de direcciones
  - Datos de procesos distintos convivan en memoria
- Objetivos:
  - Más memoria de la que físicamente tenemos
  - Protección de memoria entre procesos
  - Protección de zonas de memoria
  - Relocalización: permite que un programa se ejecute en cualquier posición de la memoria física
  - Carga rápida de procesos





# Repaso - Memoria virtual

- Algunos términos que se usan para la memoria virtual son:
  - **Página** o segmento se usan para referirse al bloque de memoria
  - **Fallos de página** o de dirección (page o address fault)
- La CPU produce direcciones virtuales que se traducen mediante una combinación de hardware y software a una dirección física, con la que se accede a la memoria principal
- A este proceso se le llama mapeo de memoria (*memory mapping*) o traducción de dirección (*address translation*)
- Los sistemas de memoria virtual se dividen en dos clases:
  - Los que utilizan bloques de tamaño fijo (**Páginas**)
  - Los que utilizan bloques de tamaño variable (**Segmentos**)
- El uso de memoria virtual paginada/segmentada afecta a la CPU:
  - El direccionamiento para memoria virtual paginada tiene una única dirección de tamaño fija dividida en n° de página y desplazamiento
  - Para la segmentada se necesitan dos palabras por dirección: una para el n° de segmento y otra para el direccionamiento dentro del segmento

# Repaso - Memoria virtual

- **Comparación cuantitativa** de la memoria caché y la memoria virtual:

| Parámetro                 | Caché de primer nivel   | Memoria virtual  |
|---------------------------|---|--|
| Tamaño de bloque (página) | 16-128 bytes  | 4096-65,536 bytes  |
| Tiempo de acierto         | 1-3 ciclos de reloj   | 100-200 ciclos de reloj  |
| Penalización por fallo    | 8-200 ciclos de reloj   | 1,000,000-10,000,000 ciclos de reloj                             |
| (tiempo de acceso)        | (6-160 ciclos de reloj)   | (800,000-8,000,000 ciclos de reloj)                              |
| (tiempo de transferencia) | (2-40 ciclos de reloj)  | (200,000-2,000,000 ciclos de reloj)                              |
| Tasa de fallos            | 0.1-10%   | 0.00001-0.001%   |
| Mapeo de direcciones      | Dirección física de 25-45 bits a dirección de caché de 14-20 bits | Dirección virtual de 32-64 bits a dirección física de 25-45 bits |

# Repaso - Memoria virtual

- **Comparación cualitativa** de la memoria caché y la memoria virtual:
  - El reemplazo en las cachés se realiza por hardware mientras que en la memoria virtual se hace mediante una combinación hardware-software controlada por el sistema operativo
  - El tamaño de la dirección que maneja el procesador determina el tamaño de la memoria virtual, pero el tamaño de la memoria caché es independiente del tamaño de esa dirección
  - En la memoria secundaria, además de las direcciones de los procesos, también se ubican los ficheros necesarios por los programas, que normalmente no forman parte del espacio de direcciones del proceso

# Repaso - Memoria virtual: las 4 preguntas

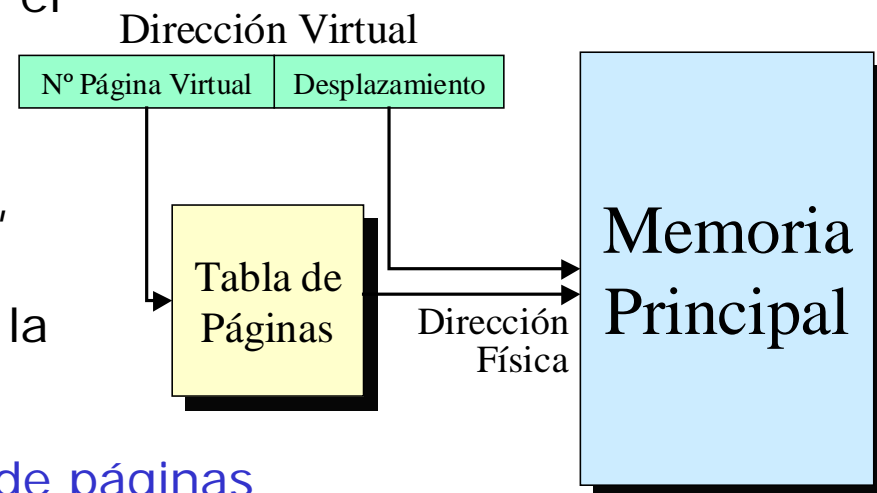
- **¿Dónde ponemos la página en memoria principal?**

- La penalización es tan alta que el SO permite a la página situarse en cualquier lugar de la memoria principal

- **¿Cómo sabemos si la página está en memoria principal?**

- **Tabla de páginas** indexada por el número de página virtual, nos da la página física

- Con direcciones de 32 bits, páginas de 4 KB y 4 bytes por entrada, el tamaño de la tabla sería de 4 MB



- Algunos SO aplican una **tabla de páginas invertida** para reducir el tamaño:

- Para el mismo caso anterior, con una memoria física de 512 MB se necesita ahora una tabla de 1 MB
- Para traducir la página rápidamente, se utiliza una pequeña caché llamada TLB

# Repaso - Memoria virtual: las 4 preguntas

- **¿Qué página deberíamos reemplazar en caso de fallo de memoria virtual?**
  - Debido a la penalización por fallo de página, el objetivo del SO es minimizar los fallos de página
  - La mayoría de los SO intentan usar la política LRU usando un bit de referencia que se limpia periódicamente
- **¿Qué ocurre en las escrituras?**
  - La penalización por acceder a disco es tan alta que es imprescindible utilizar **postescritura**
  - Las páginas se marcan con un **bit de sucio**

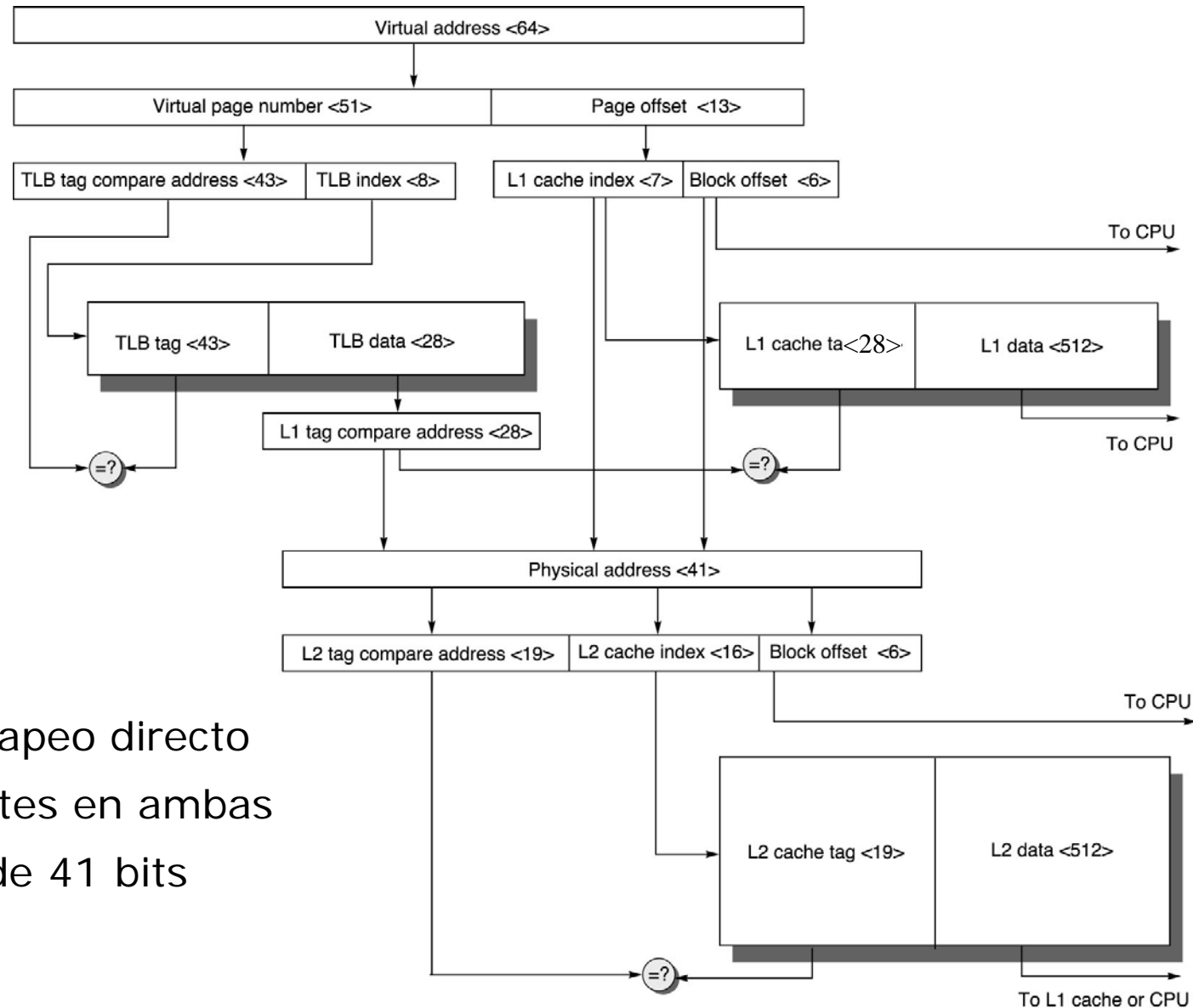
# Repaso - Memoria virtual: TLBs

## Traducción Rápida de direcciones virtuales

- Las **tablas de páginas** son **tan grandes** que deben estar en memoria principal y pueden incluso estar también paginadas:
  - Cada acceso a memoria principal cuesta dos accesos a memoria: uno para obtener la dirección física y otro para acceder a esa dirección física
- **Solución**: Caché con las últimas traducciones realizadas:
  - Esta caché **de traducciones** se llama **TLB** (*Translation Lookaside Buffer*)
  - **No reemplaza** la tabla de páginas, sólo **acelera** las traducciones
  - La etiqueta para indexar el TLB está contenida en el n° pág. virtual
  - Devuelve la **página física** y los **bits de control** de la tabla de traducción de páginas (protección, validez, uso y sucio)
  - Para cambiar la dirección física de una página o su código de protección, el SO debe forzar a que esa página salga del TLB
  - El bit de sucio que cada página tiene en el TLB significa que el contenido de la página se ha modificado, no que la dirección física haya cambiado

# Ejemplo: Jerarquía de memoria completa

- Página de 8KB
- TLB de 256 entradas de MD
- L1:
  - 8K MD unificada
  - virtualmente indexada y físicamente etiquetada
- L2 de 4MB de mapeo directo
- Bloque de 64 bytes en ambas
- Dirección física de 41 bits



# Índice

- Introducción
- Evaluación del Rendimiento de la Jerarquía de Memoria
- Reducción de la Tasa de Fallos de Caché
- Reducción de la Penalización por Fallo de Caché
- Reducción del Tiempo en Caso de Acierto en Caché
- Organizaciones de la Memoria Principal



# Evaluación del rendimiento de la jerarquía...

- A la hora de evaluar el rendimiento, ¿cómo tenemos en cuenta el número de ciclos de espera en un acceso a memoria?:

$$T_{CPU} = (N^{\circ} Ciclos_{CPU} + N^{\circ} Ciclos_{ParadaMemoria}) \times T_{Ciclo}$$

- El número de ciclos de parada de memoria depende del número de fallos y de la penalización por fallo (PF):

$$N^{\circ} Ciclos_{ParadaMemoria} = N^{\circ} Fallos \times PF = NI \times AMI \times TF \times PF$$

Donde:  $NI \equiv$  Número de instrucciones

$AMI \equiv$  Número medio de accesos a memoria por instrucción

$TF \equiv$  Tasa de fallos

- **Ejemplo:** Dada una máquina con  $CPI=1$  cuando todos los accesos a memoria aciertan en caché. Los únicos accesos a datos son los *loads* y *stores*, que representan el 50% de las instrucciones ejecutadas. Si la penalización por fallos es 25 ciclos y la tasa de fallos del 2%, ¿cuánto más rápida sería la máquina si todas las instrucciones acertasen en caché?

# Evaluación del rendimiento de la jerarquía...

- ¿Qué métrica podemos emplear para medir el rendimiento de la jerarquía de memoria?

$$T_{a\ m} = T_{s\ a} + m \times T_{p\ f}$$

$T_{a\ m}$  = Tiempo medio acceso a memoria

$T_{s\ a}$  = Tiempo de servicio en caso de acierto

$m$  = Tasa de fallos

$T_{p\ f}$  = Tiempo de penalización de fallo

- Para mejorar el rendimiento de una jerarquía de memoria, necesitamos mejorar uno o varios de los términos de la expresión anterior
- Para ello vamos a ver varias **optimizaciones** de la memoria cache organizadas en tres grandes categorías:
  - Reducción de la tasa de fallos de cache ( $m$ )
  - Reducción de la penalización en caso de fallo de cache ( $T_{p\ f}$ )
  - Reducción del tiempo de servicio en caso de acierto en cache ( $T_{s\ a}$ )

# Índice

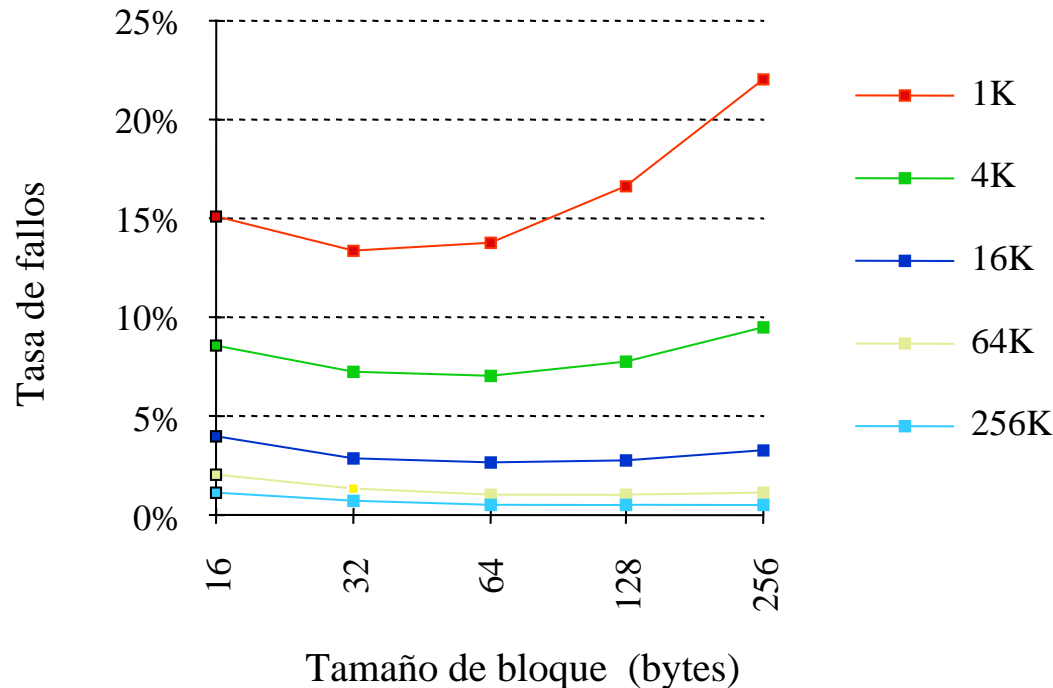
- Introducción
- Evaluación del Rendimiento de la Jerarquía de Memoria
- Reducción de la Tasa de Fallos de Caché
- Reducción de la Penalización por Fallo de Caché
- Reducción del Tiempo en Caso de Acierto en Caché
- Organizaciones de la Memoria Principal

# Clasificación de fallos de caché

- Podemos distinguir los siguientes motivos por los que se producen fallos en caché:
  - **Forzosos**<sup>/frío</sup>: el primer acceso a un bloque de memoria no puede estar en la caché (poco efecto en programas grandes)
    - Llamados fallos de arranque en frío o de primera referencia
  - **Capacidad**: la memoria caché no tiene el tamaño suficiente para contener todos los bloques necesarios
    - En un momento determinado uno de los bloques que se han utilizado tiene que dejar hueco a otro (la caché está llena)
    - Se produce fallo si se vuelve a referenciar el bloque desalojado
  - **Conflicto**: la memoria caché no es totalmente asociativa
    - Aciertos en una caché totalmente asociativa que se vuelven fallos en una asociativa por conjuntos de  $n$ -vías se deben a más de  $n$  peticiones sobre algunos conjuntos

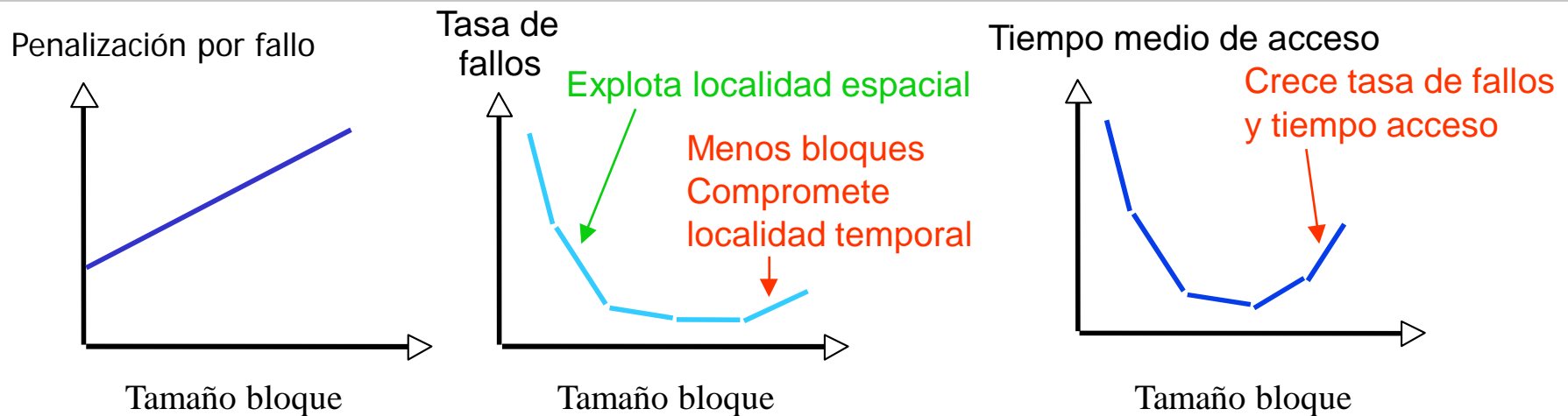
# Aumento del tamaño de bloque

- La manera más sencilla de reducir la tasa de fallos es aumentar el tamaño del bloque



- Reduce fallos forzosos, pues aumenta la localidad espacial
- Aumenta la penalización por fallo, pues cuesta más mover un bloque más grande
- Al reducir el número de bloques en caché pueden aumentar los fallos por conflicto y fallos por capacidad

# Aumento del tamaño de bloque



- El diseñador del sistema de caché debe intentar minimizar la tasa de fallos y la penalización por fallo
- La selección del tamaño del bloque depende tanto de la latencia como del ancho de banda del nivel inferior:
  - Una **gran latencia y un gran ancho de banda** aconsejan un tamaño de bloque grande, pues la caché obtiene muchos más bytes por fallo por un pequeño incremento en la penalización
  - Una **baja latencia y un bajo ancho de banda** aconsejan un tamaño de bloque pequeño, pues el tiempo que se gana respecto a un bloque grande es pequeño, y al tener un número grande de bloques pequeños se reducirán los fallos por conflictos

# Aumento de la asociatividad

## Asociativa por conjuntos 1-vía (correspondencia directa)

|   | tag | datos |
|---|-----|-------|
| 0 |     |       |
| 1 |     |       |
| 2 |     |       |
| 3 |     |       |
| 4 |     |       |
| 5 |     |       |
| 6 |     |       |
| 7 |     |       |

## Asociativa por conjuntos 2-vías

|   | tag | datos | tag | datos |
|---|-----|-------|-----|-------|
| 0 |     |       |     |       |
| 1 |     |       |     |       |
| 2 |     |       |     |       |
| 3 |     |       |     |       |

## Asociativa por conjuntos 4-vías

|   | tag | datos | tag | datos | tag | datos | tag | datos |
|---|-----|-------|-----|-------|-----|-------|-----|-------|
| 0 |     |       |     |       |     |       |     |       |
| 1 |     |       |     |       |     |       |     |       |

Hoy en día L1 4-vías y L2 8-vías o 16- vías.

## Asociativa por conjuntos 8-vía (prácticamente totalmente asociativa)

|   | tag | datos | tag | datos | tag | datos | tag | datos | tag | datos | tag | datos | tag | datos | tag | datos |
|---|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|
| 0 |     |       |     |       |     |       |     |       |     |       |     |       |     |       |     |       |

- Aumentar la asociatividad **incrementa el tiempo de servicio en caso de acierto**, pues es necesario complicar el hardware, pero también reduce la tasa de fallos
  - Hay que llegar a una solución de **compromiso**
- Hay dos heurísticas muy usadas:
  - Una caché asociativa por conjuntos de 8 vías se comporta, a efectos prácticos, como una caché totalmente 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

# Optimizaciones del compilador

- Permiten reducir la tasa de fallos sin **ningún cambio en el hardware**
- **Ejemplo:** se puede mejorar la tasa de fallos en la caché de instrucciones y en la de datos reordenando el código:
  - En 1989, McFarling observó que estudiando la información de *profiling*, podía reordenar el código para ahorrar el 50% de fallos para una caché de instrucciones de 2 KB de mapeo directo con bloques de 4 B, y un 75% en una caché de 8 KB
  - En el caso de datos se trata, si es posible, de acceder de forma consecutiva a los vecinos de bloque, de forma que se mejore la localidad espacial y temporal, y se minimicen los fallos de caché
- Vamos a ver **cuatro optimizaciones** a nivel de compilador:
  1. Combinación de arrays (*merging arrays*)
  2. Intercambio de iteraciones (*loop exchange*)
  3. Unión de bucles (*loop fusion*)
  4. *Blocking*



# Combinación de arrays (*merging arrays*)

- **Objetivo:** reducir la tasa de fallos al mejorar la localidad espacial
- Algunos programas referencian múltiples arrays en la misma dimensión, con los mismos índices al mismo tiempo
  - El problema es que estos accesos pueden interferir entre ellos provocando fallos de caché
- **Solución:** combinarlos en un único array donde cada casilla contenga los elementos necesarios de cada array original
  - De esta forma todos los datos podrían estar a la vez en un mismo bloque de caché

```
/* Antes */  
int val[SIZE];  
int key[SIZE];
```



```
/* Después */  
struct mezcla{  
    int val;  
    int key;  
};  
struct mezcla array_combinado[SIZE];
```

# Intercambio de iteraciones (*loop exchange*)

- Algunos programas tienen bucles anidados que no acceden a los datos de forma secuencial
  - Ej.: si se recorre una matriz por columnas en lugar de por filas
- **Intercambiando** el orden de los bucles podemos hacer que se accedan los datos en el orden en que están almacenados en memoria, **reduciendo los fallos de caché**
  - Se maximiza el uso de los datos de un bloque de caché antes de que éste se descarte
- Esta técnica **mejora la localidad espacial**

**/\* Antes \*/**

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



**/\* Después \*/**

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

- El código original accede a la matriz siguiendo un orden por columnas
- En la versión revisada se itera por filas, accediendo a las palabras de todo un bloque de caché antes de pasar al siguiente bloque

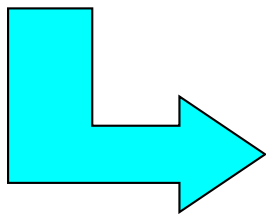
# Unión de bucles (*loop fusion*)

- Algunos programas tienen secciones de código separadas que acceden los mismos arrays, con los mismos bucles, pero realizando operaciones diferentes en los mismos datos
- Si **fusionamos** el código en un **único bucle**, los datos que se cargan en caché pueden utilizarse para las distintas operaciones antes de desalojarse
- Esta técnica **mejora la localidad temporal**

**/\* Antes \*/**

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

- En el código final, la segunda instrucción utilizará los datos que la instrucción anterior ha obligado a traer a caché



**/\* Después \*/**

```
for (i=0; i<N; i=i+1)
for (j=0; j<N; j=j+1)
{
    a[i][j] = 1/b[i][j] * c[i][j];
    d[i][j] = a[i][j] * c[i][j];
}
```

# Blocking

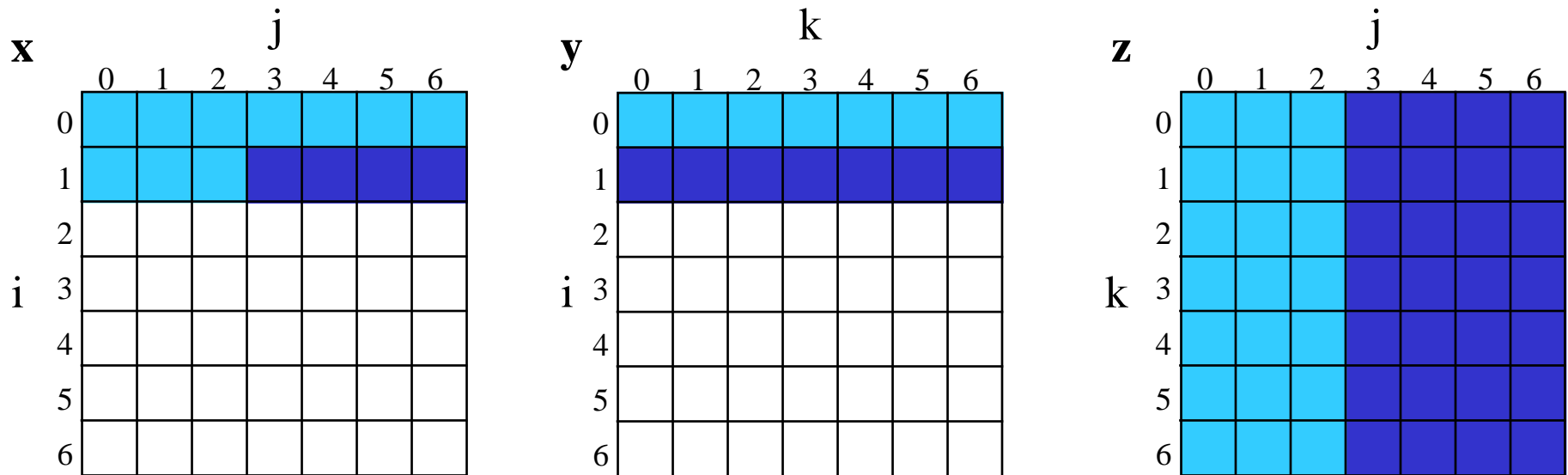
- **Objetivo:** Reducir la tasa de fallos al **mejorar la localidad temporal**
  - Útil cuando se usan varios arrays unos accedidos por filas y otros por columnas
- La idea es que en vez de operar sobre columnas o filas enteras, se opere sobre **submatrices o bloques**, maximizando el acceso a los datos de la caché antes de sustituirlos

**/\* Antes \*/**

```
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;
}
```

- Los dos bucles interiores leen todos los  $N \times N$  elementos de  $z$ , acceden repetidamente los mismos  $N$  elementos en una fila de  $y$ , y escriben una fila de  $N$  elementos de  $x$ .

# Blocking



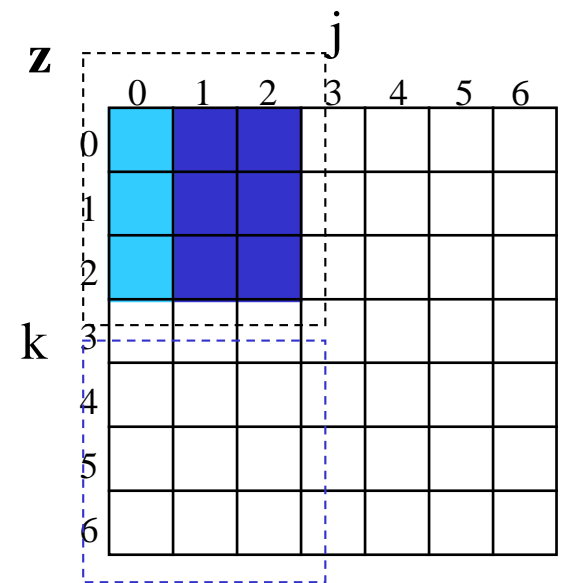
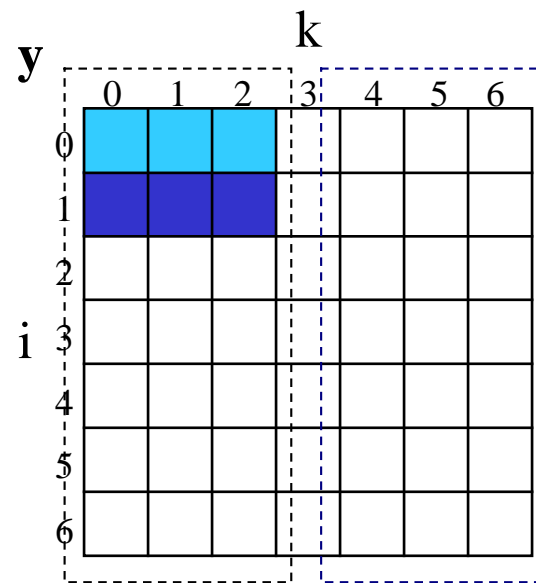
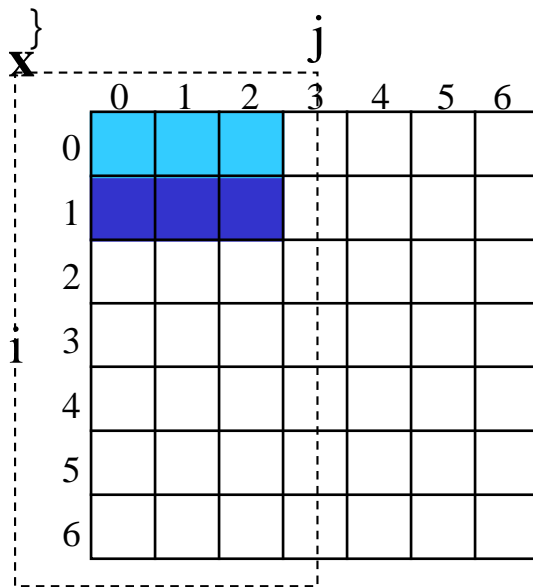
*Situación en un momento dado de los arrays: en blanco las zonas aún no accedidas, en claro las accedidas hace más tiempo, y en oscuro las accedidas recientemente.*

- El número de fallos de caché depende de  $N$  y del tamaño de la caché
- Para asegurar que el número de elementos accedidos cabe en memoria, se cambia el código original para que actúe sobre una **submatriz de tamaño  $B \times B$** , accediendo los bucles interiores en bloques de tamaño  $B$ , en vez de inicio a fin de  $x$  y  $z$
- Al tamaño  $B$  se le llama **blocking factor**

# Blocking

**/\* Después \*/**

```
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;
}
```



- Se aprovecha tanto la localidad espacial como temporal, ya que y se beneficia de la localidad espacial y z de la temporal
- Puede utilizarse también minimizar el número de *loads* y *stores*, asignando bloques a registros (siempre que el tamaño de bloque sea pequeño)

# Optimizaciones HW: caché de víctimas

- Consiste en añadir un pequeño buffer totalmente asociativo entre un nivel de caché y el siguiente donde se queden los datos que se van reemplazando en el nivel superior
- Intenta conseguir el **bajo tiempo de acierto** de la correspondencia directa a la vez que **reduce los fallos debidos a conflictos** (como las asociativas)
- Se puede acceder al mismo tiempo a la memoria caché y a la caché de víctimas, para que la penalización por fallo no se incremente
- Dependiendo de la aplicación, una caché de víctimas de cuatro entradas puede llegar a **evitar la cuarta parte** de los fallos de una caché de 4KB de mapeo directo

Funciona de forma similar a una alta asociatividad sin necesidad de tenerla

# Búsqueda anticipada por hardware de datos e instruc.

- Una solución para evitar fallos de caché es hacer que el hardware **busque anticipadamente** (*prefetching*) los datos **antes** de que los pida el procesador
  - Se trata de iniciar el acceso a memoria antes de que se ejecute una instrucción que produciría un fallo de caché
- Tanto las instrucciones como los datos pueden anticiparse, directamente en la caché o en un buffer externo (*stream buffer*) con un tiempo de acceso mucho menor que el de la memoria
- **Problemas con la prebúsqueda:**
  - Búsquedas anticipadas innecesarias → Desperdicio de ancho de banda



# Búsqueda anticipada por hardware de datos e instruc.

- La **prebúsqueda de instrucciones** se suele realizar en hardware externo a la caché:
  - Lo más típico es que el procesador busque dos bloques en cada fallo: el solicitado (que se lleva a caché) y el contiguo (que se lleva a un buffer de instrucciones)
  - Si el bloque solicitado se encuentra en el buffer de instrucciones se cancela la petición a la caché, el bloque es leído del buffer y se emite una petición de prebúsqueda para el próximo bloque

# Búsqueda anticipada por hardware de datos e instruc.

- La **prebúsqueda de datos** requiere esquemas más sofisticados:
  - Prebúsqueda con stride:
    - En lugar de prebuscar el bloque  $i+1$ , buscar  $i+x$  ( $x$  es el stride)
  - Prebúsqueda etiquetada:
    - Asociar un bit de etiqueta a cada bloque, inicialmente a 0
    - Cuando una línea es traída por fallo de caché o referenciada el bit se pone a 1
    - Cuando una línea es traída por prebúsqueda el bit se pone a 0
    - La prebúsqueda para la línea  $i+x$  se inicia cuando el bit de etiqueta de la línea  $i$  pasa de 0 a 1

# Búsqueda anticipada controlada por el compilador

- Una alternativa a la prebúsqueda hardware es dejar que el **compilador inserte instrucciones** solicitando los datos antes de que sean necesarios
- Hay dos alternativas:
  - **Register prefetch**: carga el valor en un registro (cargas normales)
  - **Caché prefetch**: carga los datos en la caché (instrucciones especiales)
- Cualquiera de las dos podría ser **faulting** o **nonfaulting**, es decir la dirección puede o no causar una excepción por un fallo en la dirección virtual y violaciones de protecciones
  - Usando esta terminología podríamos decir que una instrucción de carga usual es una *faulting register prefetch instruction*
  - La mayoría de procesadores ofrecen *nonfaulting caché prefetches*

Pre-búsqueda: cuando falla nos olvidamos

# Búsqueda anticipada controlada por el compilador

- Al usar la búsqueda anticipada por software se **incrementa el número de instrucciones**, con lo que se debe tener cuidado de que esta sobrecarga no exceda los beneficios
- Para evitar prebúsquedas innecesarias, los compiladores deben concentrarse en las referencias a memoria que tienen **mayor probabilidad de fallar** y que provocarían mayores detenciones
- La búsqueda anticipada por software sólo tiene sentido si el procesador puede continuar mientras se realiza la búsqueda (*procesador con ejecución fuera de orden*), ya que, al igual que con la búsqueda anticipada por hardware, el objetivo es que se **solape la ejecución con la búsqueda de los datos**

# Índice

- Introducción
- Evaluación del Rendimiento de la Jerarquía de Memoria
- Reducción de la Tasa de Fallos de Caché
- Reducción de la Penalización por Fallo de Caché
- Reducción del Tiempo en Caso de Acierto en Caché
- Organizaciones de la Memoria Principal

# Reducción de la penalización por fallo de caché

- La fórmula de rendimiento de la caché nos dice que **una mejora en la penalización** por fallo puede ser tan beneficiosa como la reducción de la tasa de fallos:


$$T_{a\ m} = T_{s\ a} + m \times T_p$$

- Además la velocidad de los procesadores ha crecido más rápidamente que la de las DRAMs, haciendo que el coste relativo de un fallo de caché se incremente con el tiempo
- Vamos a ver tres optimizaciones posibles para reducir la penalización en caso de fallo:
  1. Cachés multinivel
  2. Buffer de escritura
  3. Cachés no bloqueantes

# Cachés multinivel

- La gran diferencia en prestaciones entre procesador y memoria hace que nos interesen las siguientes opciones para la caché:
  - **Cachés más rápidas** para cumplir con la velocidad de la CPU
  - **Cachés más grandes** para evitar tener que ir a memoria
- **Solución:** **añadir un segundo nivel** de caché entre la caché original y memoria
  - La caché de **primer nivel** puede ser lo suficientemente **pequeña** para ser casi tan rápida como el procesador
  - La caché de **segundo nivel** puede ser lo suficientemente **grande** para capturar la mayoría de los accesos que irían a memoria principal, **disminuyendo así la penalización por fallo**
- Aunque el concepto de añadir otro nivel de caché en la jerarquía de memoria es sencillo, complica el análisis de prestaciones

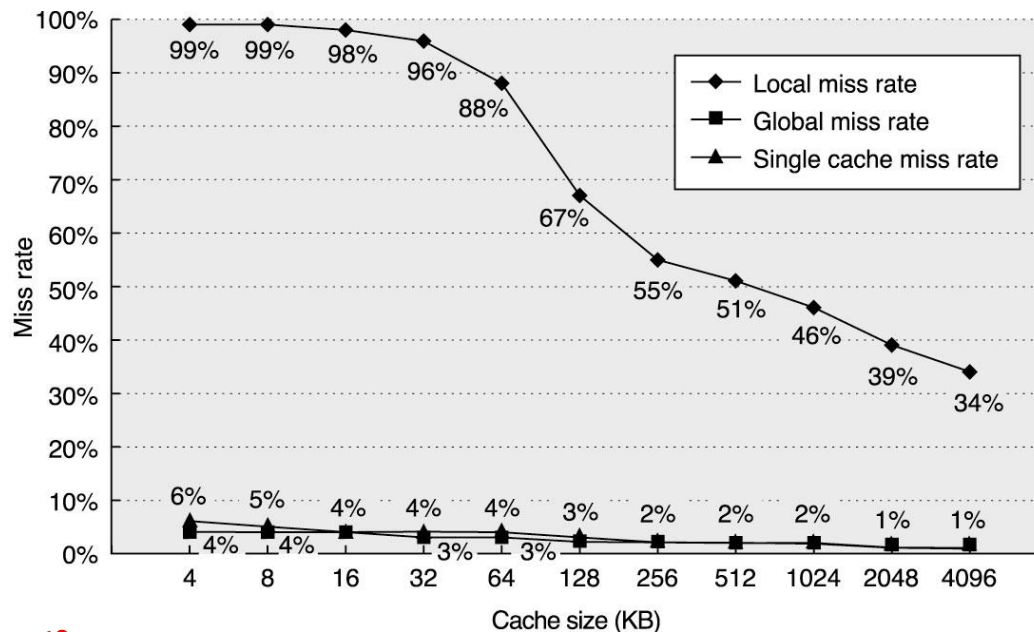
# Cachés multinivel

- Sabemos calcular el tiempo de acceso medio a memoria, pero ahora tendremos que distinguir entre los distintos niveles de caché:  

- Para evitar ambigüedad, se adoptan los siguientes términos:
  - **Tasa de fallos local** (*local miss rate*): el número de fallos en la caché dividido por el número total de accesos a esa caché. Para la caché de segundo nivel sería
  - **Tasa de fallos global** (*global miss rate*): el número de fallos de caché dividido por el número total de accesos a memoria generados por la CPU. Para una caché de 2 niveles sería
- La tasa de fallos local para la de segundo nivel es mayor ya que la caché de primer nivel captura la mayoría de accesos a memoria
- La tasa de fallos global **es más útil** e indica qué fracción de los accesos a memoria que hace la CPU van a memoria principal

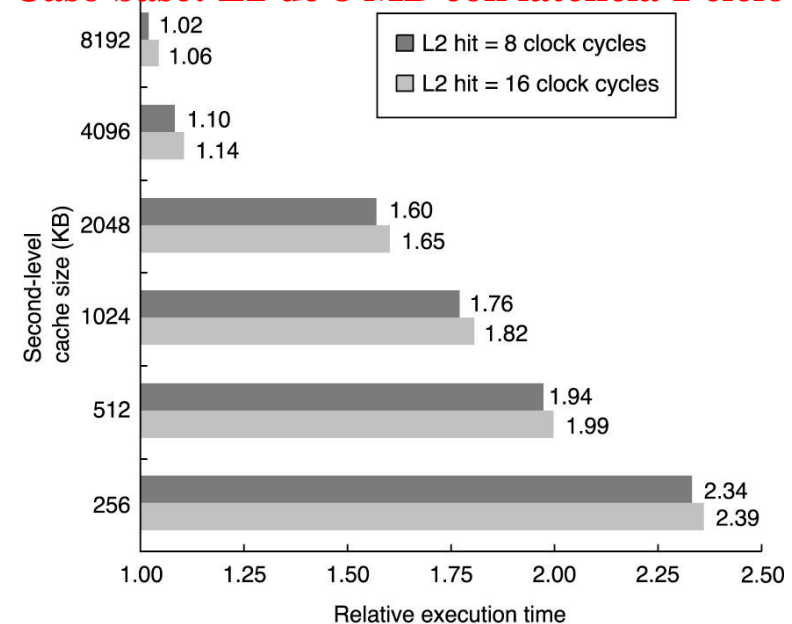


# Cachés multinivel

- Como varía **la tasa de fallos y el tiempo de ejecución relativo** en función del tamaño de caché para un sistema con uno o dos niveles de caché (el tamaño de la L1 es 32 KB):



## Caso base: L2 de 8 MB con latencia 1 ciclo



## Conclusiones

- La tasa de fallos global es muy similar a la tasa de fallos de un único nivel con el tamaño de la L2
- La tasa de fallos local no es una buena medida para la L2
- El tiempo de acierto en la L2 no es tan importante como un tamaño suficiente

# Cachés multinivel: sobre el diseño de la L2

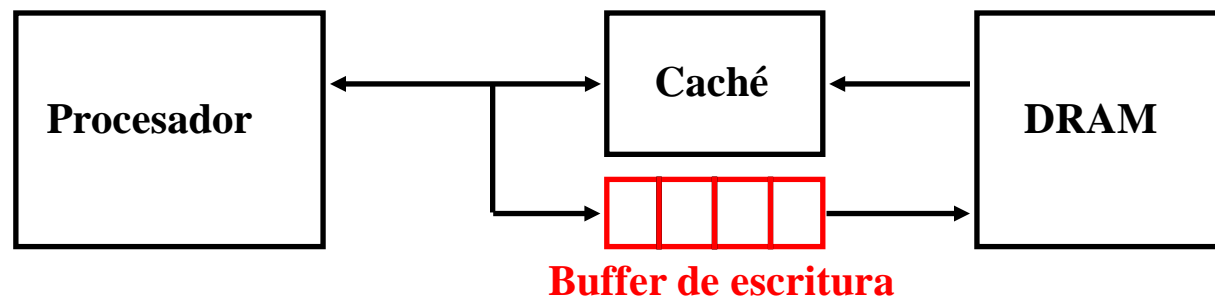
- La principal diferencia entre los dos niveles de caché es que la velocidad de la L1 afecta al tiempo de ciclo de la CPU, mientras que la de la L2 afecta a la penalización por fallo de la L1
- Dos posibilidades:
  - **Inclusión Multinivel:** L1 está contenida en L2 ( $L2 \uparrow\uparrow\uparrow L1$ )
    - La coherencia entre cachés puede determinarse mirando solamente en la L2
    - Podemos usar bloques más pequeños para L1 y mayores para L2
    - Ejemplo: P4 tiene bloques de 64 bytes en L1 y de 128 bytes en la L2. Hay que invalidar todos los bloques de la L1 que mapean en el bloque de la L2
  - **Exclusión Multinivel:** L2 no contiene a L1 ( $L2 \approx L1$ )
    - No se permite tener en L2 una copia un dato que esté en L1
    - Un fallo en L1 provoca un intercambio de bloques entre L1 y L2 en lugar de reemplazarlo
    - Ejemplo: AMD Athlon (L1 de 64 KB y L2 de 256 KB)
- La L1 y la L2 se diseñan como un todo. Ej: L1 WT y L2 WB

# Cachés multinivel: reducir tasa de fallo de la L2

- Un mayor grado de **asociatividad** en la L2 es muy interesante:
  - Incrementaría el tiempo de servicio en caso de acierto de la L2 ...
  - ... sin embargo hemos visto que para la L2 lo más importante es reducir la tasa de fallos para evitar tener que ir a memoria
- Aumentar el **tamaño** de la caché de segundo nivel:
  - Reduce los fallos por conflicto al distribuir los datos entre más bloques
  - Elimina muchos de los fallos de capacidad
- Incrementar el **tamaño del bloque** de la L2:
  - Aumentaría los fallos por conflicto para las cachés de segundo nivel pequeñas ...
  - ... pero como las cachés L2 son bastante grandes, es posible tener tamaños de bloque para la L2 de 64, 128 o 256 bytes
  - A mayor tamaño de bloque ⇒ mayor penalización por fallo

# Buffer de escritura

- Utilizar un **buffer de escritura**:
  - Las escrituras dejan el dato en el buffer, en lugar de esperar a que se escriba en memoria, de forma que las siguientes instrucciones (incluidas otros accesos a memoria) puedan seguir ejecutándose
  - Su uso es imprescindible en cachés de **escritura directa**
    - Sin él, todas las escrituras provocarían largas detenciones
  - Permite que la caché siga atendiendo otros accesos mientras realiza una escritura en memoria (o en el siguiente nivel de caché)



# Buffer de escritura

- Tener un buffer de escritura también supone una mejora para una caché con **post-escritura** acelerando los reemplazos:
  - Supongamos un fallo de lectura que va a provocar el desalojo de un bloque de caché marcado como cambiado (bit **dirty** = 1). La lectura debería esperar a que se escribiera en memoria principal el bloque desalojado
  - Sin embargo podríamos tener un buffer donde copiar el bloque modificado mientras que se trae de memoria principal el bloque demandado por la lectura para que el procesador pueda continuar
  - Una vez terminada la lectura, se copia el bloque modificado desde el buffer intermedio a memoria principal
  - De forma similar al caso con escritura directa, habría que controlar las posibles lecturas sobre un bloque que estuviera en ese buffer intermedio

# Buffer de escritura

- El buffer de escritura puede contener valores pendientes para un bloque que se ha pedido
- **Ejemplo:** Vamos a fijarnos en esta secuencia de código:

```
SW 512(R0), R3      ; M[512] ← R3
LW R1, 1024(R0)     ; R1 ← M[1024]
LW R2, 512(R0)      ; R2 ← M[512]
```

Caché de mapeo directo con escritura directa y direcciones 512 y 1024 en el mismo hueco (buffer de escritura de 4 palabras)

- Se produce un riesgo RAW a través de memoria
- Tras el SW, el dato de R3 se sitúa en el buffer de escritura
- El siguiente LW usa el mismo índice de caché ⇒ se produce fallo
- El 2º LW intenta poner el valor de la posición 512 en el registro R2, lo que también provoca fallo de caché
- Si el buffer de escritura no ha completado la escritura en la posición 512 de memoria, el 2º LW podría leer de memoria el valor viejo

# Buffer de escritura

- La forma más sencilla de solucionar este problema es que la **lectura se espere hasta que se vacíe el buffer de escritura**, pero esto incrementaría el tiempo de penalización en caso de fallo para la lectura:
  - Los diseñadores del MIPS M/1000 estimaron que para un buffer de escritura de 4 palabras, si cada lectura que provocara un fallo de caché se esperara a que se vaciara el buffer, se incrementaría la penalización en caso de fallo en un factor de 1.5
- La alternativa es que cuando se produzca un fallo por una lectura **se compruebe** el buffer de escritura y si no está ahí el dato buscado y el sistema de memoria está disponible, se continúe con la lectura en memoria principal
  - Prácticamente todos los procesadores usan esta solución, dando prioridad a las lecturas sobre las escrituras

# Cachés no bloqueantes

- Hasta ahora un acceso a memoria que produce **un fallo de caché detiene el cauce** hasta que se obtiene la palabra que lo provoca:
  - Un procesador que permite terminación fuera de orden no necesita parar cuando hay un fallo en la caché de datos
  - Por ejemplo, la CPU continuaría buscando instrucciones de la caché de instrucciones mientras la caché de datos resuelve el fallo
- Las **cachés no bloqueantes** (*lockup-free cache*) permiten que la caché de datos siga permitiendo accesos mientras resuelve un fallo de caché de otra instrucción
- Puede permitir sólo accesos que acierten (**Acierto bajo fallo**) o incluso solapar varios fallos (**Acierto bajo múltiples fallos**), lo que reduciría la penalización media por fallo
  - Incrementa la complejidad del controlador de caché, pues puede haber varios accesos al mismo tiempo a memoria
  - Solo es beneficiosa si la memoria puede servir varios fallos a la vez



# Índice

- Introducción
- Evaluación del Rendimiento de la Jerarquía de Memoria
- Reducción de la Tasa de Fallos de Caché
- Reducción de la Penalización por Fallo de Caché
- Reducción del Tiempo en Caso de Acierto en Caché
- Organizaciones de la Memoria Principal

# Acceso segmentado a la caché

- El tiempo de servicio en caso de acierto para la L1 es un valor crítico pues afecta directamente a la frecuencia de la CPU
- Los procesadores modernos tienen **segmentado el acceso a las cachés** (instrucciones y datos) a lo largo de varios ciclos para poder soportar una elevada frecuencia de reloj:
  - Ejemplos:
    - Pentium III, 2 ciclos para acceder a caché de instrucciones
    - Pentium 4 consume 4 ciclos
  - Aumenta el número de etapas del cauce, provocando mayor penalización en caso de fallo de predicción de saltos y más ciclos entre la entrada de una carga y el uso del dato
  - **Aumenta el ancho de banda** de memoria, pero no disminuye la latencia de un acierto en caché

# Cachés más sencillas y pequeñas

- Una parte importante del tiempo que se tarda en un acierto en caché se gasta en leer la etiqueta y compararla con la dirección.
- En general, hardware más pequeño significa más rápido, por lo que interesa que la L1 sea **pequeña**
  - Así mismo, es crítico mantener al menos una L2 on-chip suficientemente grande para evitar la penalización de tener que salir fuera del chip
- A la vez, sería interesante que la caché fuera lo más **sencilla** posible para tener un tiempo de acierto pequeño
  - Por ejemplo, en una caché de mapeo directo se puede solapar la lectura de la etiqueta con la de los datos, lo cual es muy interesante para una L1
- La necesidad de conseguir altas frecuencias sin incrementar mucho el número de etapas del cauce hace que las L1 sean pequeñas y sencillas

# Cachés más sencillas y pequeñas (cont.)

- Aunque la cantidad de caché que se integra dentro del chip del procesador va incrementándose a lo largo del tiempo, la tendencia actual es a tener la misma cantidad de L1:
  - Tres generaciones de los microprocesadores de AMD (K6, Athlon y Opteron) tienen el mismo tamaño de L1
- Se está poniendo énfasis en aumentar la frecuencia de reloj y ocultar los fallos de la L1 mediante ejecución dinámica (**mientras el fallo se resuelve se ejecutan otras instrucciones del programa**) y el uso de L2 para evitar tener que ir a memoria

# Evitar la traducción de la dirección al indexar la caché

- Incluso con una caché pequeña y sencilla se debe hacer la **traducción** de la dirección virtual que maneja la CPU a la dirección física para acceder a memoria
- Una alternativa es usar la **dirección virtual** para la caché, con lo que nos evitamos tener que hacer la traducción: **cachés virtuales** frente a las cachés tradicionales que llamaremos **cachés físicas**
- Es importante distinguir la comparación de las etiquetas (**etiquetas virtuales vs. físicas**) y la indexación de la caché (**direcciones virtuales vs. físicas**)
- Si usar direcciones virtuales evita tener que esperar a que se traduzca la dirección, ¿por qué no se usan siempre cachés virtuales?

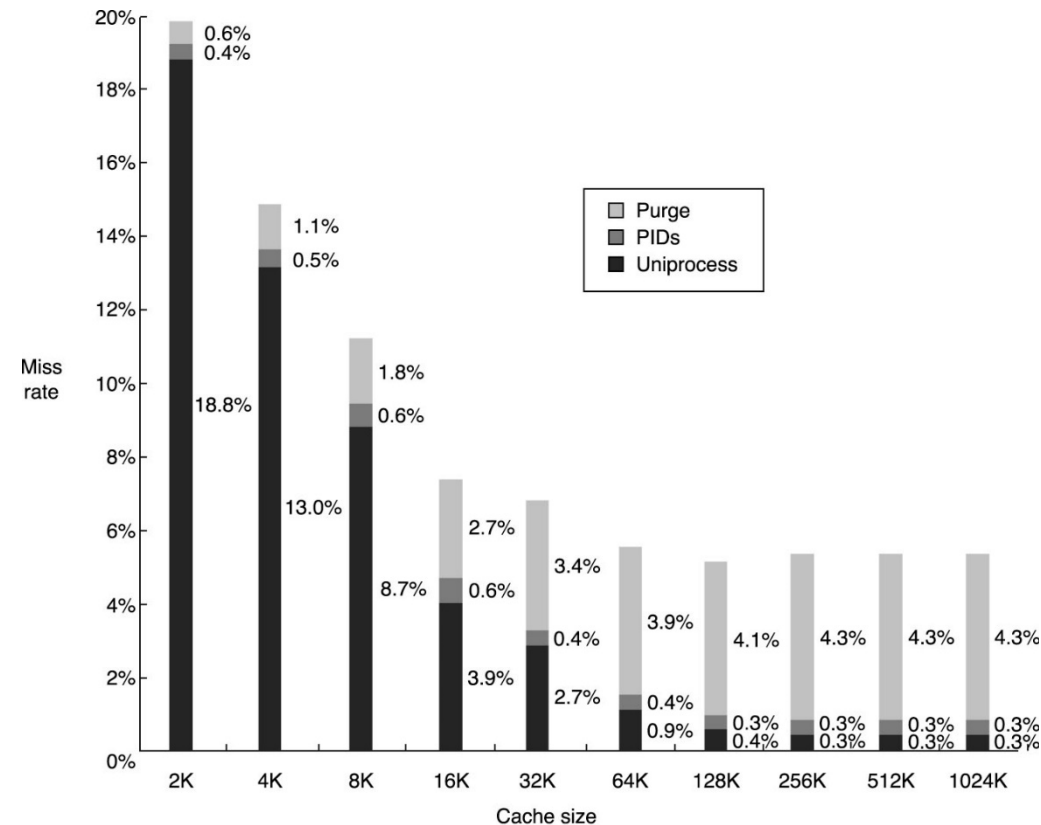
# Problemas de las cachés virtuales

- **Problema 1: Comprobar la protección de memoria**
- Al acceder a memoria se comprueban los permisos (**a nivel de página**) durante la traducción de dirección virtual a física
  - Esta información se guarda en la tabla de páginas
- **Solución:** Copiar la información sobre protección de la tabla de páginas en la caché en caso de fallo y comprobarla en cada acceso a la caché:
  - Cada vez que hay un fallo de caché se lleva a la misma el bloque de datos y la información de protección
  - Cada vez que hay una acierto se comprueban los permisos en la caché

# Problemas de las cachés virtuales

## • Problema 2: Cambios de contexto

- En cada cambio de contexto, las mismas direcciones virtuales se refieren a distintas direcciones físicas, por tanto la caché debe vaciarse
- Añadir a la etiqueta un campo PID que asigna el SO, de forma que sólo necesita vaciar la caché cuando se reutilice un PID



# Problemas de las cachés virtuales

- **Problema 3: Sinónimos o alias**

- Se suelen utilizar diferentes direcciones virtuales para la misma dirección física para compartir zonas de memoria (sinónimos o alias) → **Varias copias del mismo dato en caché**
- Si se modifica uno, el otro podría mantener su valor antiguo, provocando incoherencias



# Cachés indexadas virtualmente y etiquetadas físicamente

|           |               |              |
|-----------|---------------|--------------|
| Nº Página | Offset página |              |
| Etiqueta  | Índice        | Despl. Block |

- En estas cachés el **campo índice** de la caché ha de caer dentro del campo *offset* de página de la **dirección virtual**
- Esto permite simultanear el acceso (indexación) a la caché con la traducción de la etiqueta a su dirección física. Finalmente, la comparación de etiquetas se hace con direcciones físicas
- **Problema:**
  - El campo índice no puede exceder el campo offset. Esto se traduce en una limitación en el tamaño efectivo de las cachés
- **Solución:** Aumentar la asociatividad para mantener el tamaño del índice:



- Así conseguimos que doblando la asociatividad tengamos una **caché el doble de grande** y sin aumentar su campo índice

# Resumiendo: tiempos de acceso según el tipo de caché

## 1. Cachés físicas

- 1º: acceso TLB → traducción
- 2º: acceso caché → dato

$$T_{am} = T_{am\_TLB} + T_{am\_cache} =$$

$$= T_{sa\_TLB} + m_{TLB}PF + T_{sa_{L1}} + m_{L1}PF$$

## 2. Cachés virtuales

- 1º: acceso caché → dato
- Y solo se accede al TLB **en caso de fallo** pues se necesita la traducción para acceder a memoria principal:

$$T_{am} = T_{sa_{L1}} + m_{L1}[PF + (T_{sa_{TLB}} + m_{TLB}PF)]$$

## 3. Cachés index. virtualmente y etiq. físicamente

- Se accede a ambas, caché y TLB, **en paralelo**.  
Asumiendo que  $T_{sa}$  del TLB es menor que  $T_{sa}$  de la L1, quedaría:

$$T_{am} = T_{sa_{L1}} + m_{L1}PF + m_{TLB}PF$$

# Índice

- Introducción
- Evaluación del Rendimiento de la Jerarquía de Memoria
- Reducción de la Tasa de Fallos de Caché
- Reducción de la Penalización por Fallo de Caché
- Reducción del Tiempo en Caso de Acierto en Caché
- Organizaciones de la Memoria Principal

# Organizaciones de la memoria principal

- La memoria principal satisface las peticiones de las cachés y sirve como interfaz para las operaciones de E/S ya que los datos se suelen transferir desde y hacia la memoria
- Las métricas para medir el rendimiento de la memoria son:
  - **Latencia**: fundamental para las cachés, ya que afecta a la penalización por fallo de caché
  - **Ancho de banda**: fundamental para el subsistema de E/S
- El empleo de cachés **de segundo nivel** en prácticamente todos los sistemas y los **tamaños de bloque bastante grandes** que usan, hace que el **ancho de banda** entre memoria principal y caché sea importante
  - Uno de los motivos por los que los diseñadores incrementan el tamaño de bloque de la caché es para aprovechar el elevado ancho de banda de la memoria
- Vamos a estudiar distintas organizaciones de memoria principal para **incrementar el ancho de banda**

# Organizaciones de la memoria principal

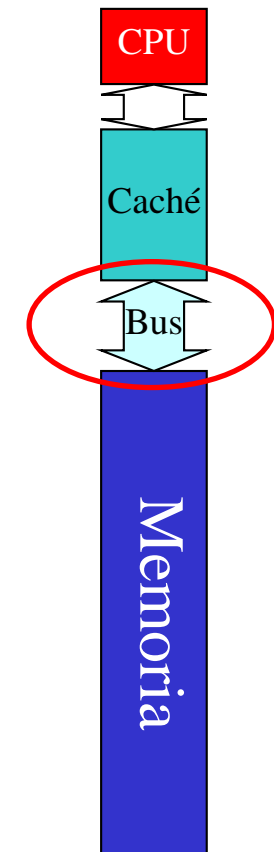
- **Ejemplo:** supongamos una caché con tamaño de bloque de 4 palabras (cada una de 8 bytes) con las siguientes características:
  - 4 ciclos para enviar la dirección
  - 56 ciclos de tiempo de acceso
  - 4 ciclos para mandar una palabra de datos
- Para la configuración de la figura tendríamos:
  - Tiempo de penalización por fallo

$$T_{PF} = 4 \times (4 + 56 + 4) = 256 \text{ ciclos}$$

- Ancho de banda de memoria

$$BW = \frac{32}{256} = 0.125 \text{ bytes / ciclo}$$

Organización de memoria de una palabra de anchura



# 1ª Técnica: Mayor anchura

- Podemos **aumentar la anchura de la caché y la anchura del bus** que conecta caché y memoria con el fin de aumentar el **ancho de banda** entre ambas

- Con una anchura de **dos palabras** tenemos:
  - Tiempo de penalización en caso de fallo

$$T_{PF} = 2 \times (4 + 56 + 4) = 128 \text{ ciclos}$$

- Ancho de banda

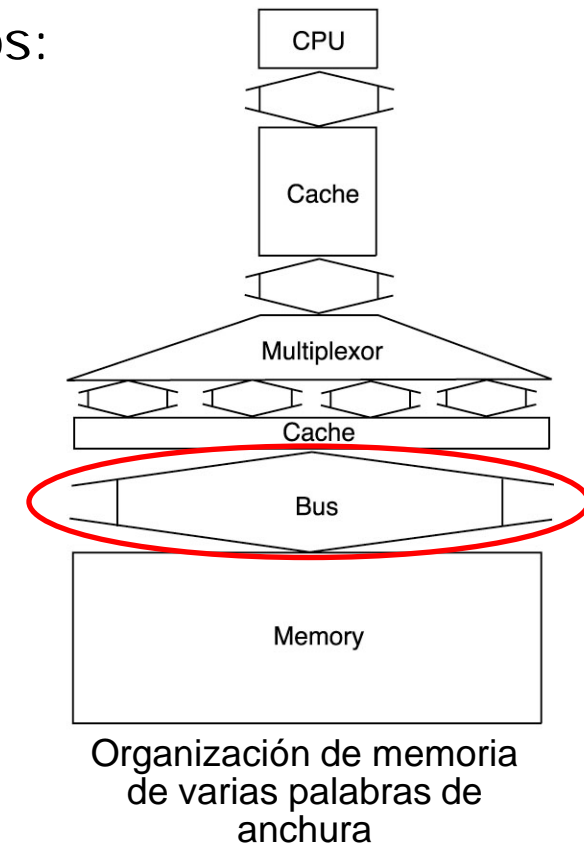
$$BW = \frac{32}{128} = 0.25 \text{ bytes / ciclo}$$

- Con una anchura de **cuatro palabras**:
  - Tiempo de penalización en caso de fallo

$$T_{PF} = 4 + 56 + 4 = 64 \text{ ciclos}$$

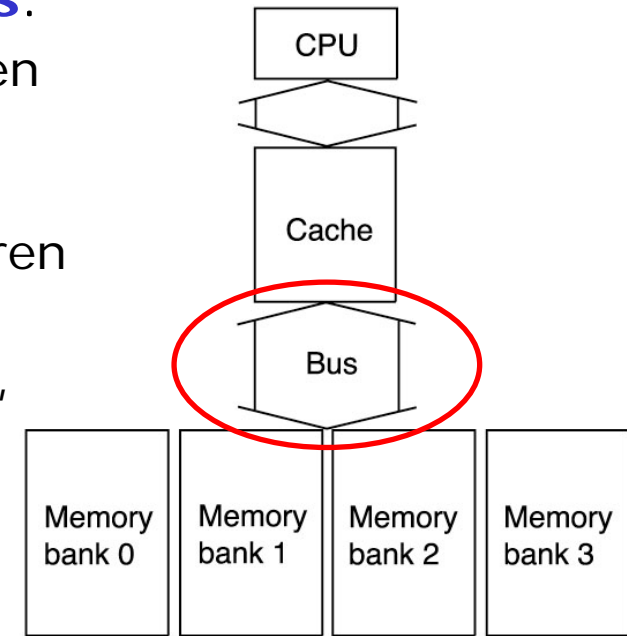
- Ancho de banda

$$BW = \frac{32}{64} = 0.5 \text{ bytes/ciclo}$$



## 2ª Técnica: Memoria entrelazada

- Podemos organizar la memoria en **bancos**:
  - Direcciones **consecutivas** se almacenan en **bancos diferentes**
  - Podemos leer/escribir varias palabras simultáneamente siempre que se encuentren en diferentes bancos
  - Los bancos tienen anchura de una palabra, con lo que **no es necesario cambiar la anchura del bus ni de la caché**
- Enviando una dirección a 4 bancos ahora tenemos:
  - Tiempo de penalización en caso de fallo:



$$T_{PF} = 4 + 56 + (4 \times 4) = 76 \text{ ciclos}$$

- Ancho de banda

$$BW = \frac{32}{76} = 0.42 \text{ bytes/ciclo}$$

## 2ª Técnica: Memoria entrelazada

- Esta técnica es menos costosa que la anterior y proporciona resultados parecidos:
  - Antes:** anchura de 4  $\rightarrow$  BW=0.5
  - Ahora:** anchura de 1  $\rightarrow$  BW=0.42
- La memoria se entrelaza normalmente con un “**factor de entrelazado**” de una palabra con lo que se optimizan los accesos secuenciales

| Word address | Bank 0 | Word address | Bank 1 | Word address | Bank 2 | Word address | Bank 3 |
|--------------|--------|--------------|--------|--------------|--------|--------------|--------|
| 0            |        | 1            |        | 2            |        | 3            |        |
| 4            |        | 5            |        | 6            |        | 7            |        |
| 8            |        | 9            |        | 10           |        | 11           |        |
| 12           |        | 13           |        | 14           |        | 15           |        |



### 3ª Técnica: Bancos de memoria independientes

- La memoria entrelazada utiliza bancos que trabajan con las mismas líneas de dirección **compartiendo el controlador de memoria**
- Una generalización del entrelazado es permitir **múltiples accesos independientes** gracias a tener múltiples controladores de memoria:
  - Cada banco necesita **líneas de dirección**, y posiblemente de datos, **separadas**
  - Se puede acceder al mismo tiempo a **direcciones diferentes** en cada banco
- Las **cachés no bloqueantes** sólo tienen sentido si tenemos **bancos independientes**