



# ESTRUCTURA DE COMPUTADORES

## Tema 2. Memoria cache

Dpto. Arquitectura de Computadores y Automática  
Universidad Complutense de Madrid



- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



## ⊙ **Bibliografía**

- ⊙ John L. Hennessy & David A. Patterson , “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann, 5ª ed. Capítulo 2, Apéndice B
- ⊙ David A. Patterson & John L. Hennessy, “Computer Organization and Design. The Hardware/Software Interface”, Morgan Kaufmann 5ª ed. Capítulo 5



- ⊙ **Introducción: Jerarquía de memoria**
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos

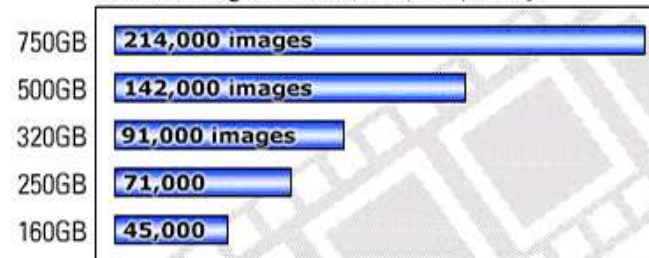


# INTRODUCCIÓN

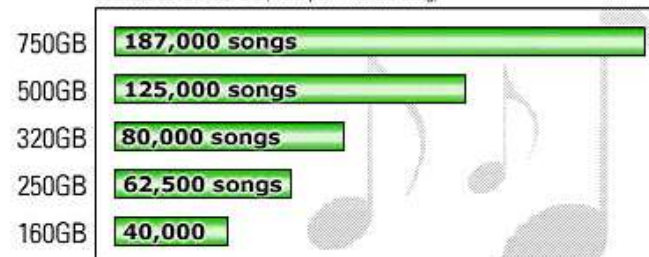
**Hours of High Definition Video** (Digital video - 8.3GB per hour)



**Number of Digital Photos** (3.5MB per 6 Mpixel image)



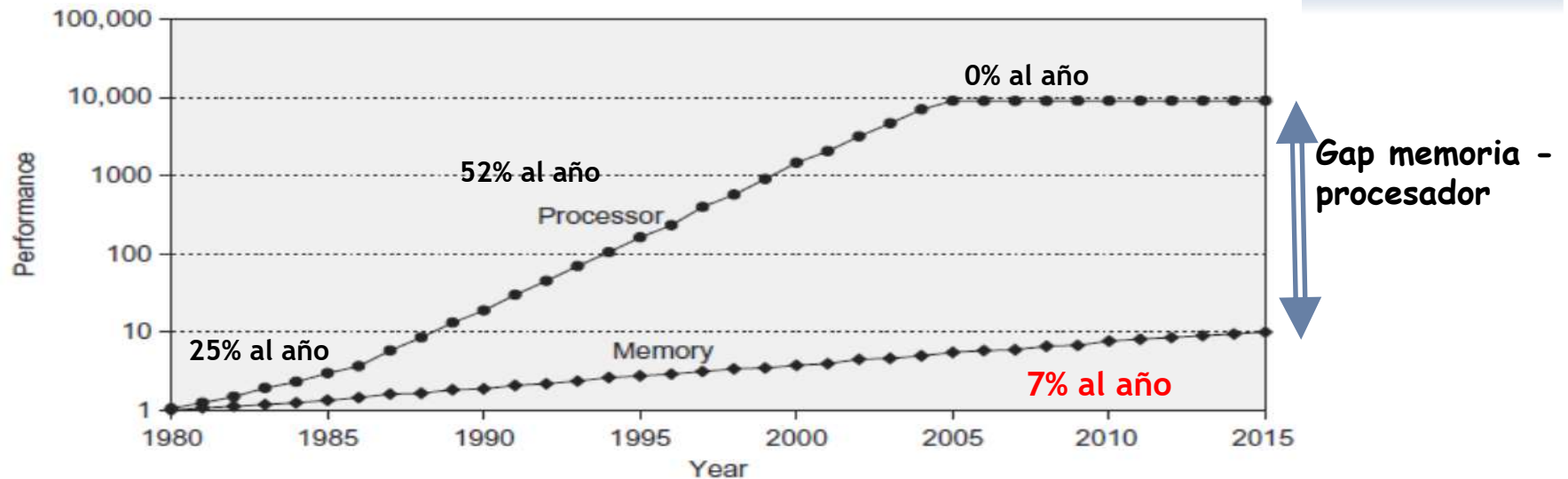
**Number of MP3s** (4MB per 4 minute song)



- ⦿ Los datos ocupan espacio en memoria
- ⦿ Acceder a ellos implica un cierto tiempo



# INTRODUCCIÓN: EL PROBLEMA



- © La demanda de anchura de banda con memoria crece
  - © Segmentación, ILP
  - © Ejecución especulativa
  - © 1980 no caches “on chip”, 2020 3 niveles de cache “on chip”



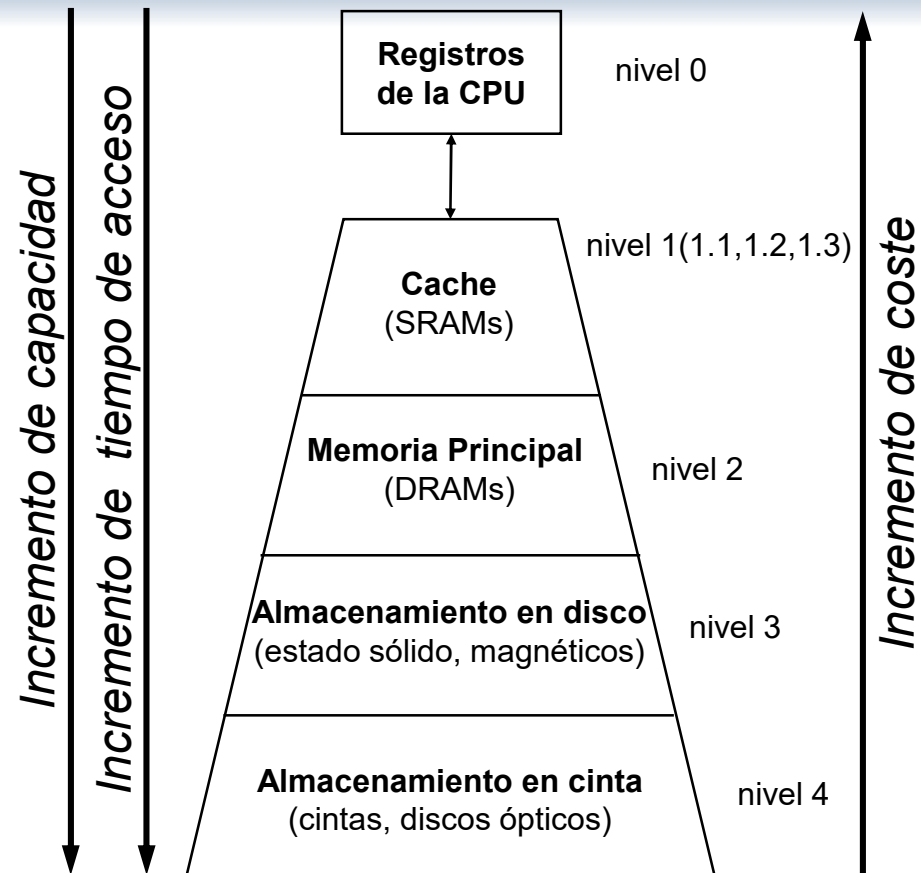
# INTRODUCCIÓN: TECNOLOGÍAS DISPONIBLES

Tipo de memoria (2015)	Tiempo de acceso (ns)	\$ por Gbyte	Ancho de Banda (Gbytes/s)
SRAM	0.5	1000	25+
DRAM	10-50	40	10
Disco de Estado Sólido (SSD)	20.000	2	0,5
Disco magnético	5.000.000-20.000.000	0.09	0,75



# JERARQUÍA DE MEMORIA

- ⊙ Un computador típico está formado por diversos niveles de memoria, organizados de forma jerárquica:
  - ⊙ Registros de la CPU
  - ⊙ Memoria Cache
  - ⊙ Memoria Principal
  - ⊙ Memoria Secundaria (discos)
  - ⊙ Unidades de Cinta (Back-up) y Dispositivos Ópticos
- ⊙ El coste de todo el sistema de memoria excede al coste de la CPU
  - ⊙ Es muy importante optimizar su uso







## ⊙ **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 de 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 (caches), 2 (memoria principal) y 3 (memoria secundaria)
  - 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)



# JERARQUÍA DE MEMORIA

## © Gestión de la memoria cache

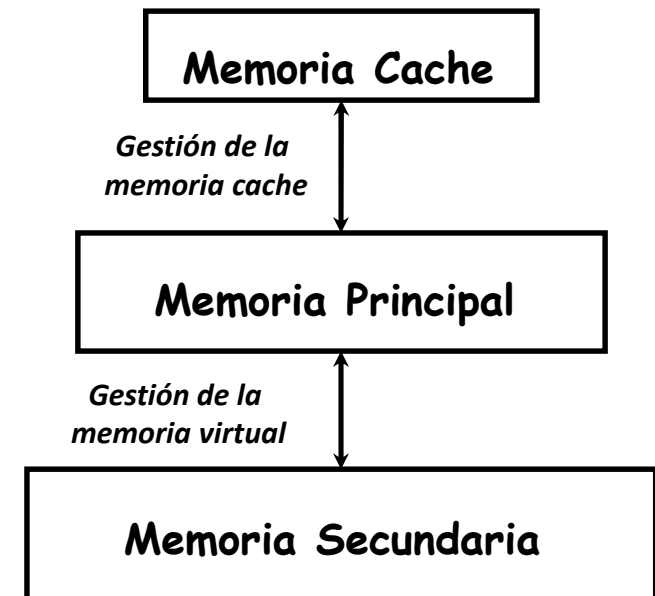
- © Controla la transferencia de información entre la memoria cache y la memoria principal
- © Suele llevarse a cabo mediante Hardware específico (MMU o “Memory Management Unit”)

## © 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 Sistema Operativo

## © Objetivo:

- Conseguir una memoria de gran tamaño, rápida y al menor coste posible
- De forma transparente al usuario





# PROPIEDADES DE LA JERARQUÍA DE MEMORIA

## ⊙ Inclusión

- ⊙ Cualquier información almacenada en el nivel de memoria  $M_i$ , debe encontrarse también en los niveles  $M_{i+1}$ ,  $M_{i+2}$ , ...,  $M_n$ .

$$M_1 \subset M_2 \subset \dots \subset M_n$$

## ⊙ Coherencia

- ⊙ Las copias de la misma información existentes en los distintos niveles deben ser coherentes
  - Si un bloque de información se modifica en el nivel  $M_i$ , deben actualizarse los niveles  $M_{i+1}, \dots, M_n$



# PROPIEDADES DE LA JERARQUÍA DE MEMORIA

## ⦿ Localidad

- ⦿ Las referencias a memoria generadas por la CPU, para acceso a datos o a instrucciones, están concentradas o agrupadas en ciertas regiones del tiempo y del espacio

## ⦿ Localidad temporal

- Las direcciones de memoria (instrucciones o datos) recientemente referenciadas, serán referenciadas de nuevo, muy probablemente, en un futuro próximo
- Ejemplos: Bucles, subrutinas, accesos a pila, variables temporales, etc.

## ⦿ Localidad espacial

- Tendencia a referenciar elementos de memoria (datos o instrucciones) cercanos a los últimos elementos referenciados
- Ejemplos: programas secuenciales, arrays, variables locales de subrutinas, etc.



## JERARQUÍA DE MEMORIA

- ⊙ **Bloque:** unidad mínima de transferencia entre dos niveles
  - ⊙ En cache es habitual llamarle “línea” y en memoria virtual “página” o “segmento”
- ⊙ **Acierto (hit):** el dato solicitado está en el nivel i
  - ⊙ Tasa de aciertos (*hit ratio*): la fracción de accesos encontrados en el nivel i
  - ⊙ Tiempo de acierto (*hit time*): tiempo de detección del acierto + tiempo de acceso del nivel i. (Tiempo total invertido para obtener un dato cuando éste se encuentra en el nivel i)
- ⊙ **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 (*miss penalty*): tiempo invertido para mover un bloque del nivel i+1 al nivel i, cuando el bloque referenciado no está en el nivel i.
- ⊙ Requisito: Tiempo de acierto  $\ll$  Penalización de fallo



# GESTIÓN DE LA JERARQUÍA DE MEMORIA

- ⊙ 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 direcciona 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



- ⊙ Introducción: Jerarquía de memoria
- ⊙ **Memoria cache**
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



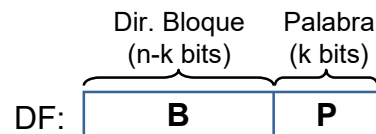
# MEMORIA CACHE

## ⊙ Estructura del sistema memoria cache/principal:

### ⊙ *MP (memoria principal):*

- formada por  $2^n$  palabras direccionables “dividida” en  $nB$  bloques de tamaño fijo de  $2^k$  palabras por bloque

Campos de una dirección física:

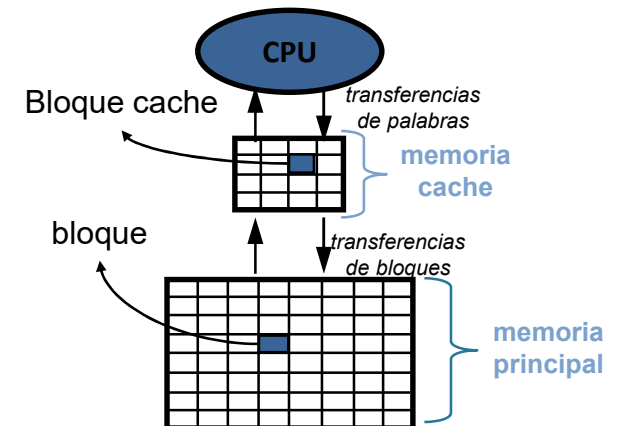


### ⊙ *MC (memoria cache):*

- formada por  $nM$  bloques (o líneas) de  $2^k$  palabras cada uno ( $nM \ll nB$ )

### ⊙ *Directorio (en memoria cache):*

- Para cada bloque de MC, indica cuál es el bloque de MP que está alojado en él



$nB$ : número de bloques de memoria

$nM$ : número de bloques de cache

$B$ : dirección del bloque

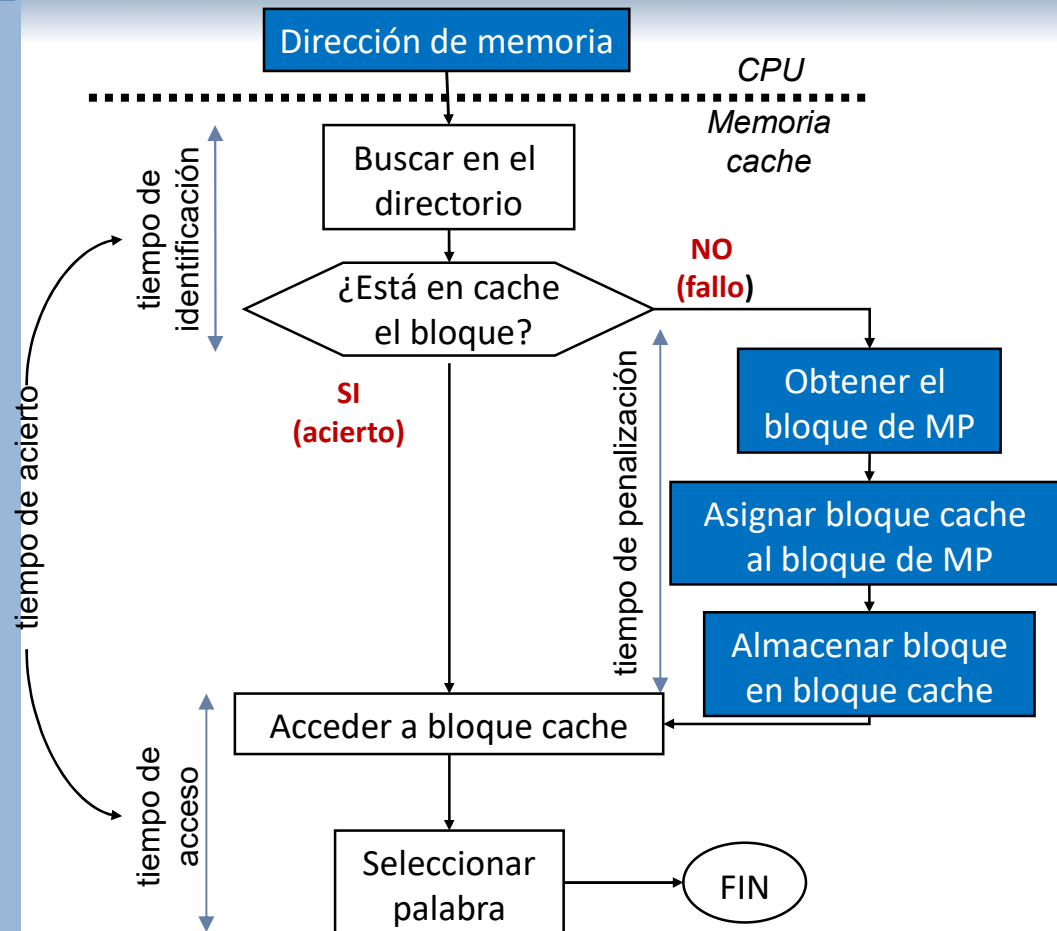
$M$ : dirección del bloque de cache

$P$ : palabra dentro del bloque





# MEMORIA CACHE



## Objetivos:

- Maximizar la tasa de aciertos
- Minimizar el tiempo de acceso
- Minimizar el tiempo de penalización
- Reducir el coste hardware

$$T_{total} = T_{acierto} + (1 - H) * T_{penalizacion}$$

(frecuencia de aciertos)



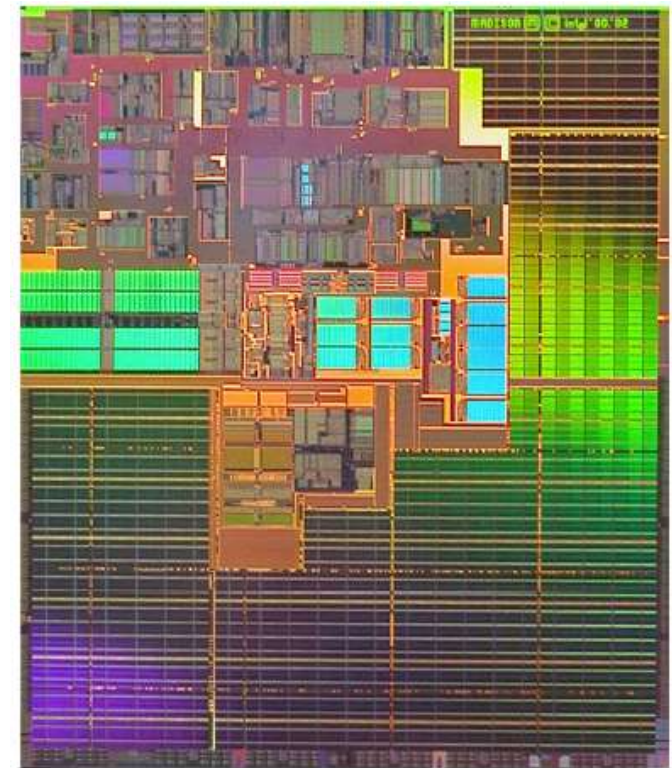
# MEMORIA CACHE: EVOLUCIÓN

Tamaño de la cache  
Del 50 al 75 % del área. Más del 80% de los transistores



**PentiumIII**

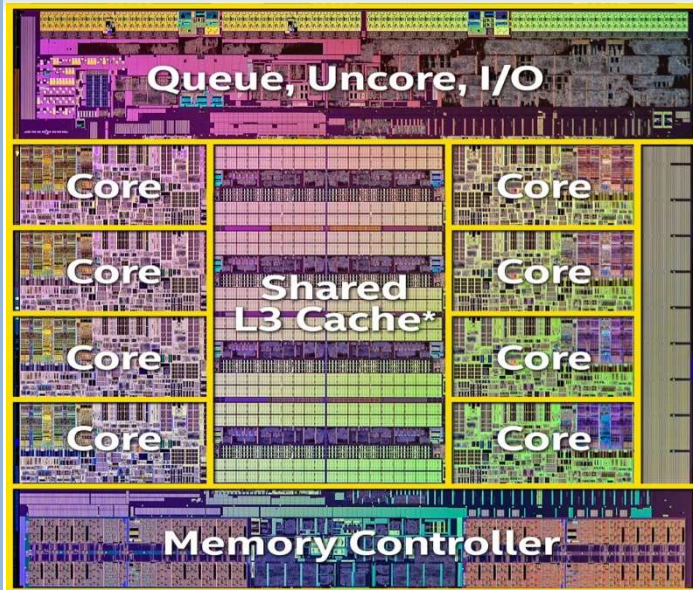
Latencia:  
1ciclo ( Itanium2) a 3 ciclos Power4-5



**Itanium 2  
Madison**

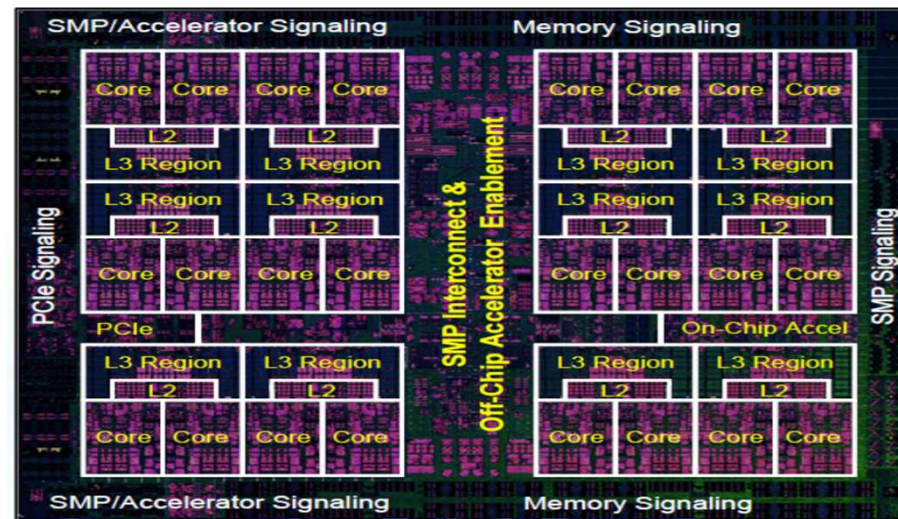


# MEMORIA CACHE: EVOLUCIÓN



Intel core i7 5960X  
L1datos 32KB, L1 instrucciones 32KB  
L2 256KB  
L3 20MB

IBM Power 9 ( 24 cores) total 128 MB  
L1 datos 32KB, L1 instrucciones 32KB  
L2 256KB  
L3 120MB





- ⊙ Introducción: Jerarquía de memoria
- ⊙ **Memoria cache**
  - ⊙ **Políticas de emplazamiento**
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos

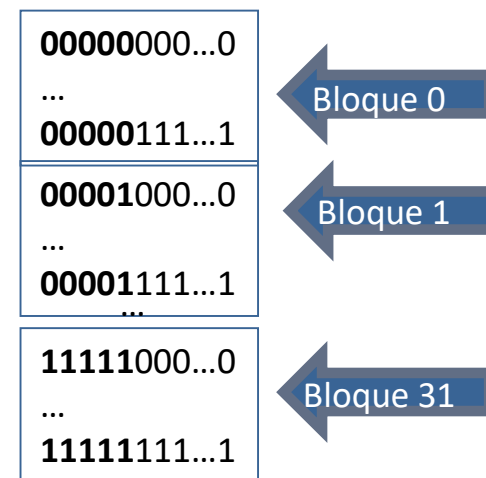
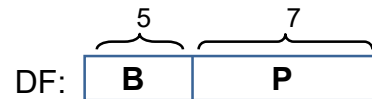


- ⊙ ¿Cómo sabemos que un dato está en la cache?, y si está, ¿cómo lo encontramos?
- ⊙ **Política de emplazamiento:**
  - ⊙ Determina en qué bloque, o bloques, de MC, puede cargarse cada bloque de MP
- ⊙ Existen diferentes políticas:
  - ⊙ Emplazamiento directo
  - ⊙ Emplazamiento asociativo
  - ⊙ Emplazamiento asociativo por conjuntos



# POLÍTICAS DE EMPLAZAMIENTO

- ⊙ Para todos los ejemplos:
  - ⊙ Tamaño de bloque 128 palabras  $\Rightarrow k=7$
  - ⊙ Memoria cache con 8 bloques
  - ⊙ Memoria principal 4k palabras  $\Rightarrow n = 12$

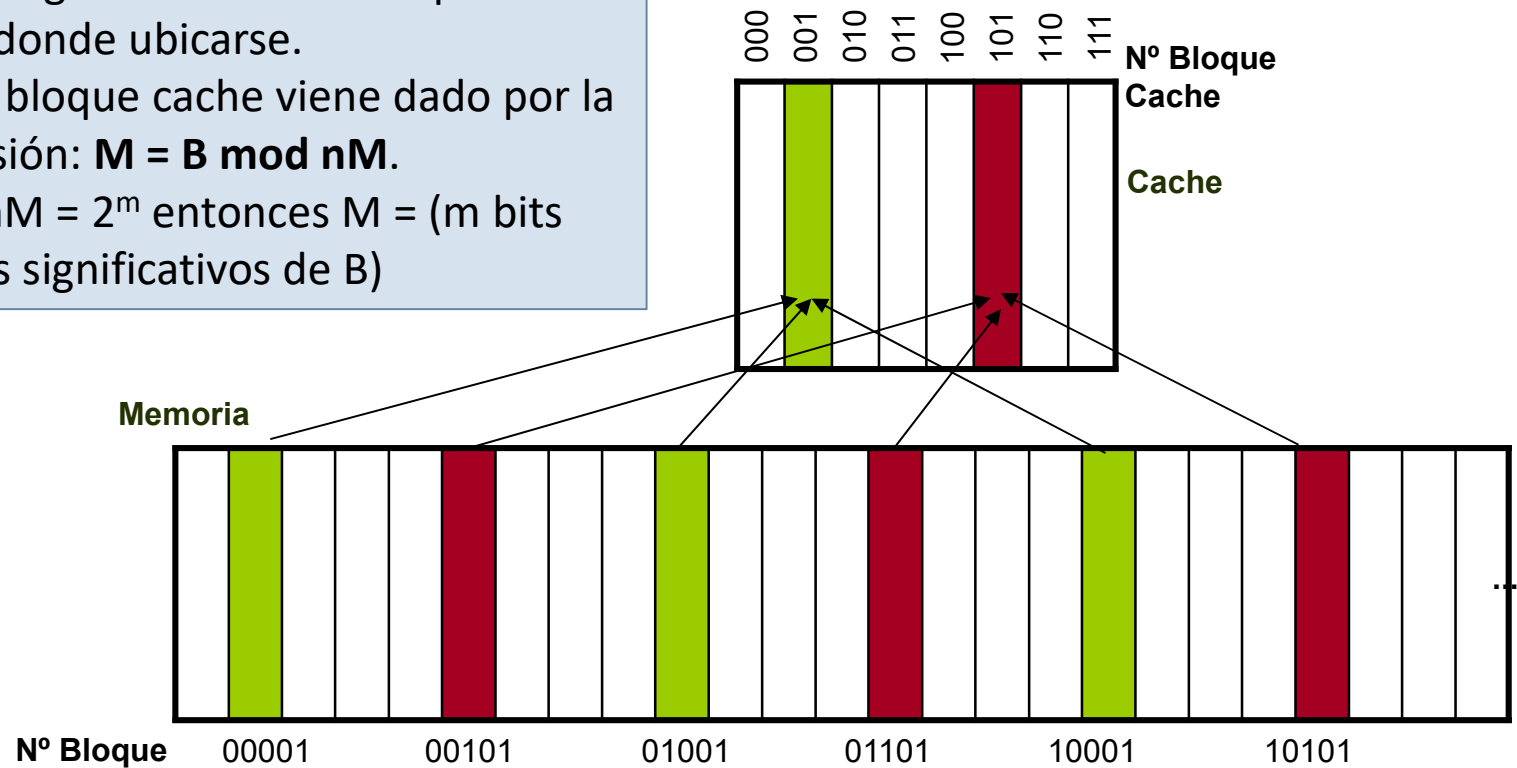






# EMPLAZAMIENTO DIRECTO

- Cada bloque B de memoria principal tiene asignado un **único** bloque cache M en donde ubicarse.
- Este bloque cache viene dado por la expresión:  $M = B \bmod nM$ .
- Si  $nM = 2^m$  entonces  $M = (m \text{ bits menos significativos de } B)$

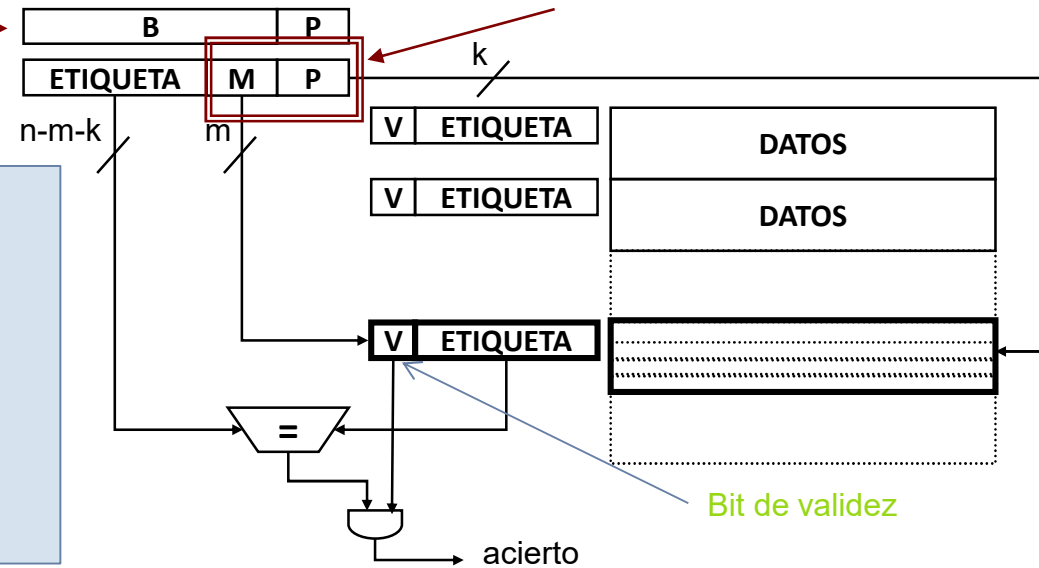




# EMPLAZAMIENTO DIRECTO

- El acceso al bloque cache y al directorio es directo.
- Para conocer si un bloque de memoria principal está cargado en MC, basta con comparar las etiquetas

Dirección ofrecida por la CPU



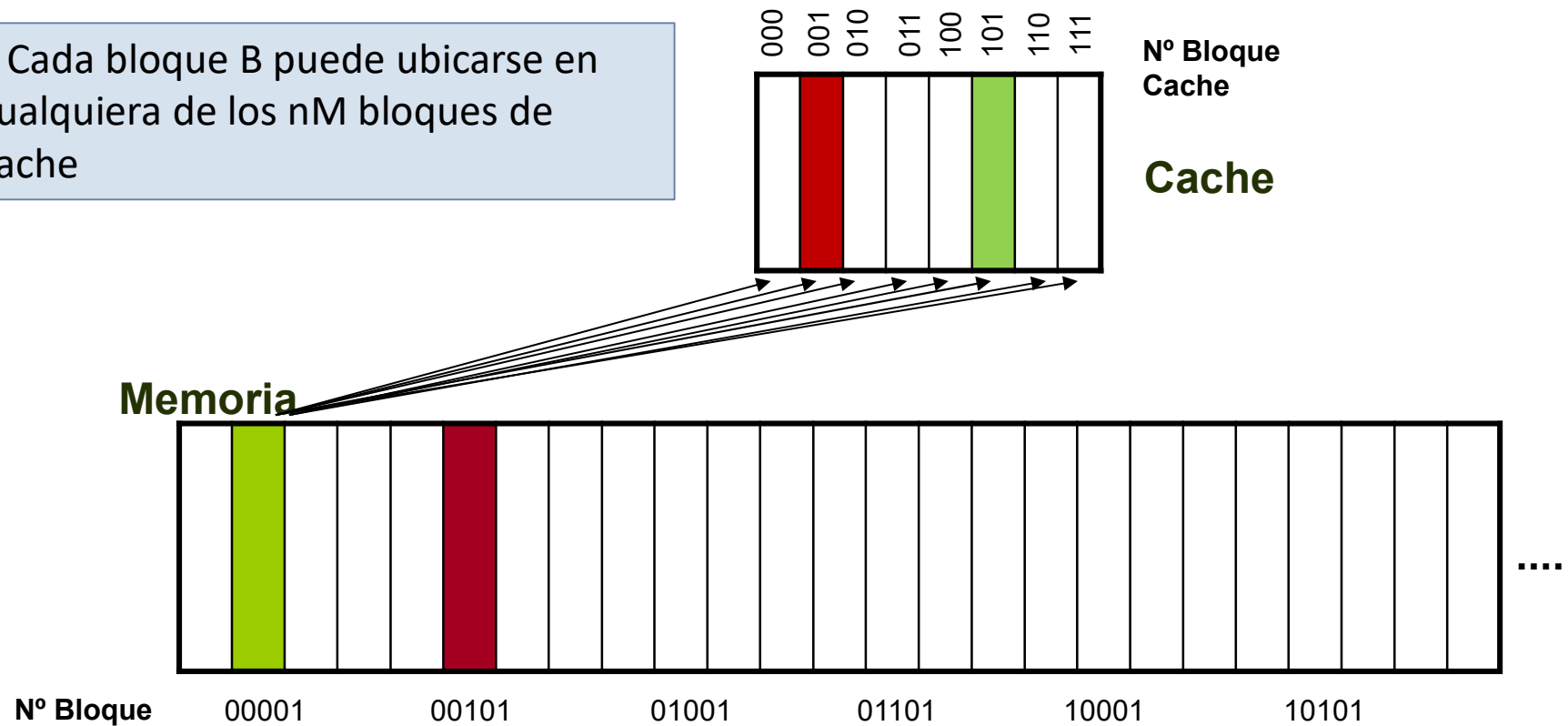
¿Cuánto bits tiene  $m$  para los datos del ejemplo?





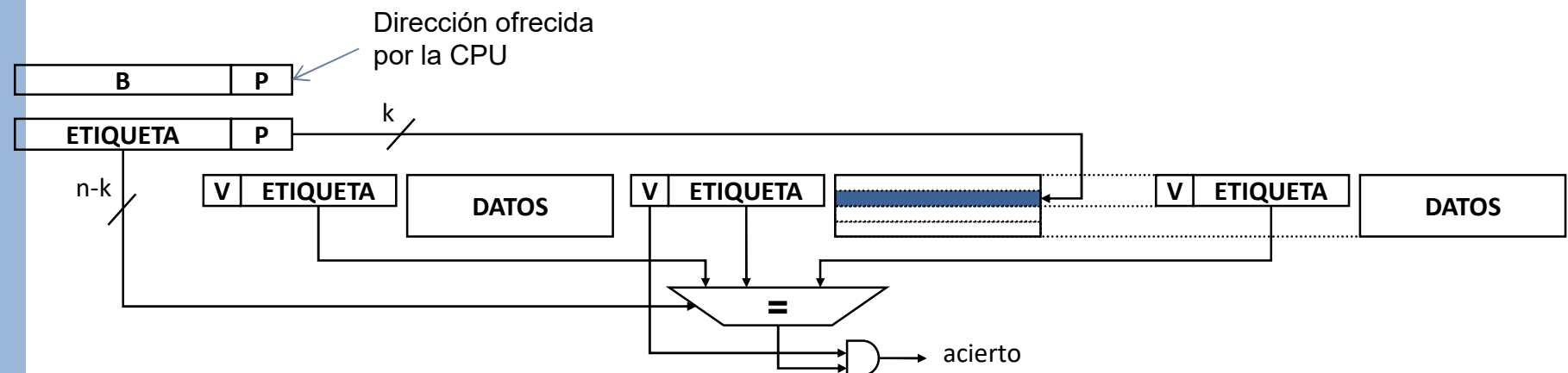
# EMPLAZAMIENTO ASOCIATIVO

- Cada bloque B puede ubicarse en cualquiera de los  $nM$  bloques de cache





# EMPLAZAMIENTO ASOCIATIVO

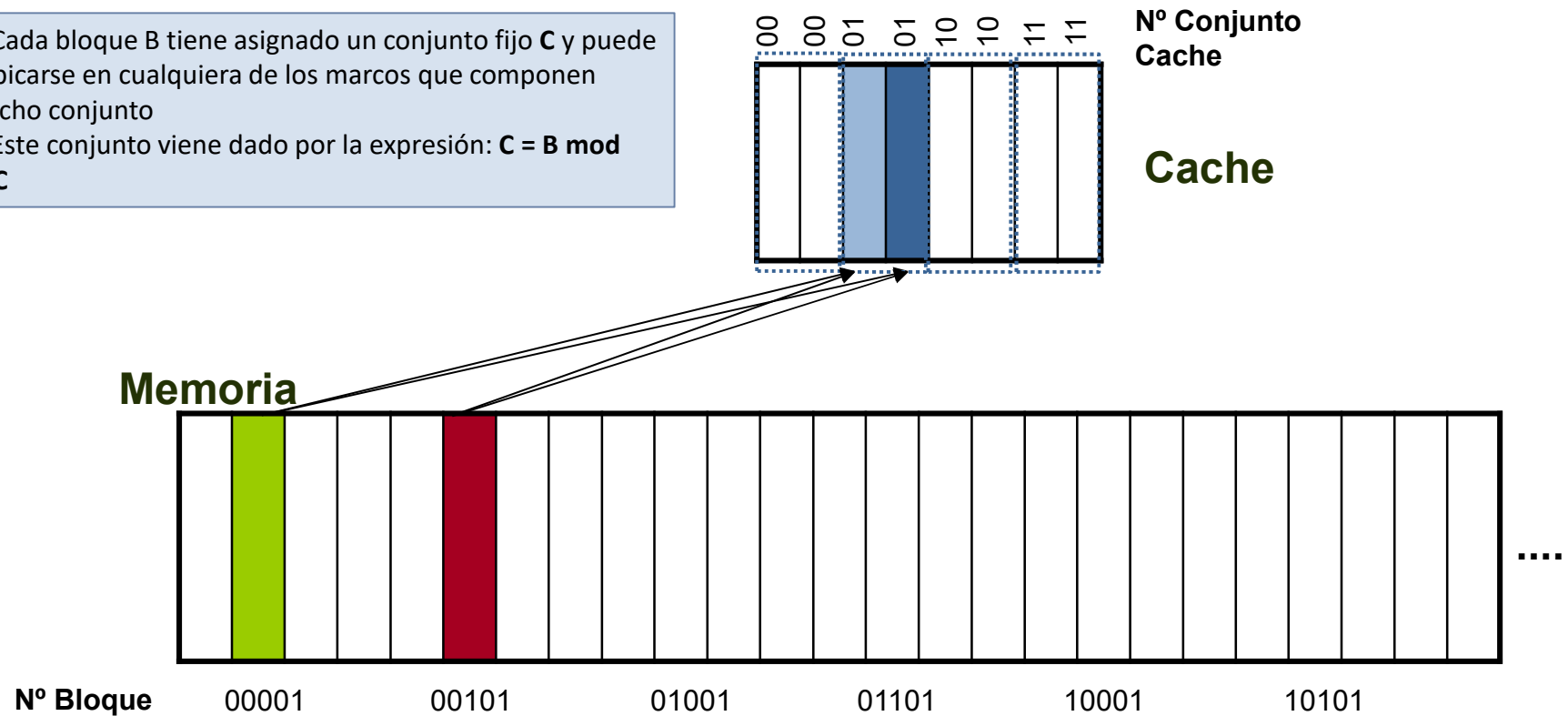


- Para conocer si un bloque de memoria principal está cargado en MC, hay que comparar todas las etiquetas



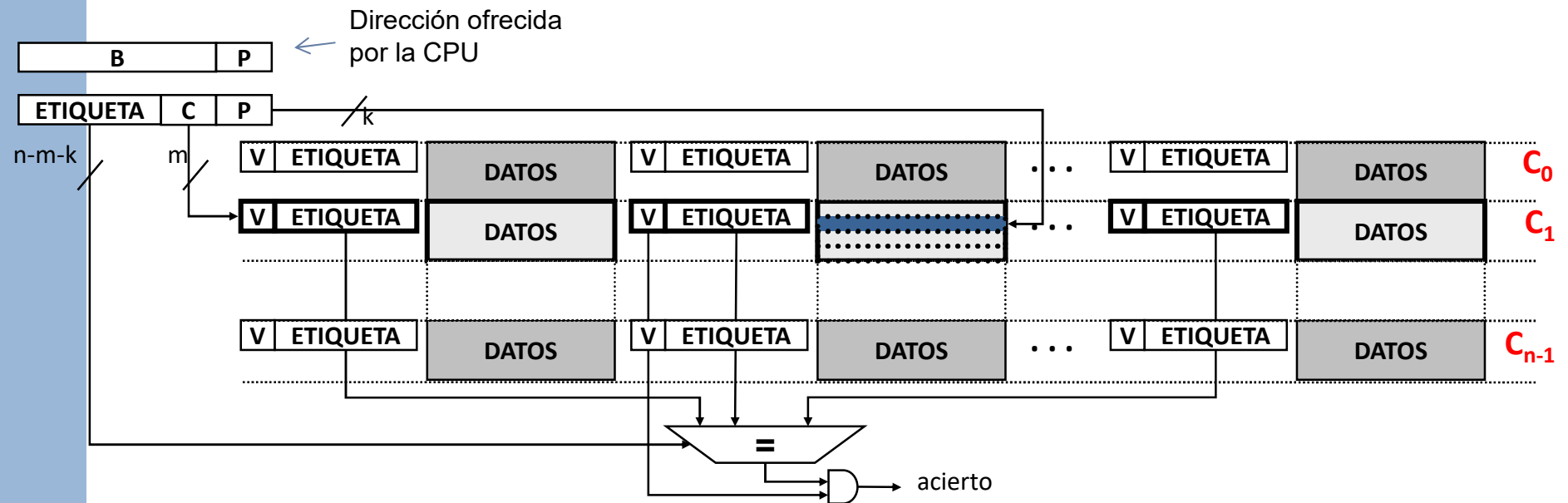
# EMPLAZAMIENTO ASOCIATIVO POR CONJUNTOS

- Cada bloque B tiene asignado un conjunto fijo C y puede ubicarse en cualquiera de los marcos que componen dicho conjunto
- Este conjunto viene dado por la expresión:  $C = B \bmod nC$





# EMPLAZAMIENTO ASOCIATIVO POR CONJUNTOS



- El directorio almacena para cada marco una etiqueta con los  $n-k-m$  bits que completan la dirección del bloque almacenado
- El acceso al conjunto es directo y al marco dentro del conjunto asociativo

¿Cuánto vale  $m$  para los datos del ejemplo si hay 4 conjuntos?



## EMPLAZAMIENTO ASOCIATIVO POR CONJUNTOS

- ⊙ El valor  $nM/nC$  se denomina grado de asociatividad o número de vías (ways) de la MC:
  - ⊙ Grado de asociatividad = 1, equivale a emplazamiento directo
  - ⊙ Grado de asociatividad =  $nM$ , equivale a emplazamiento asociativo



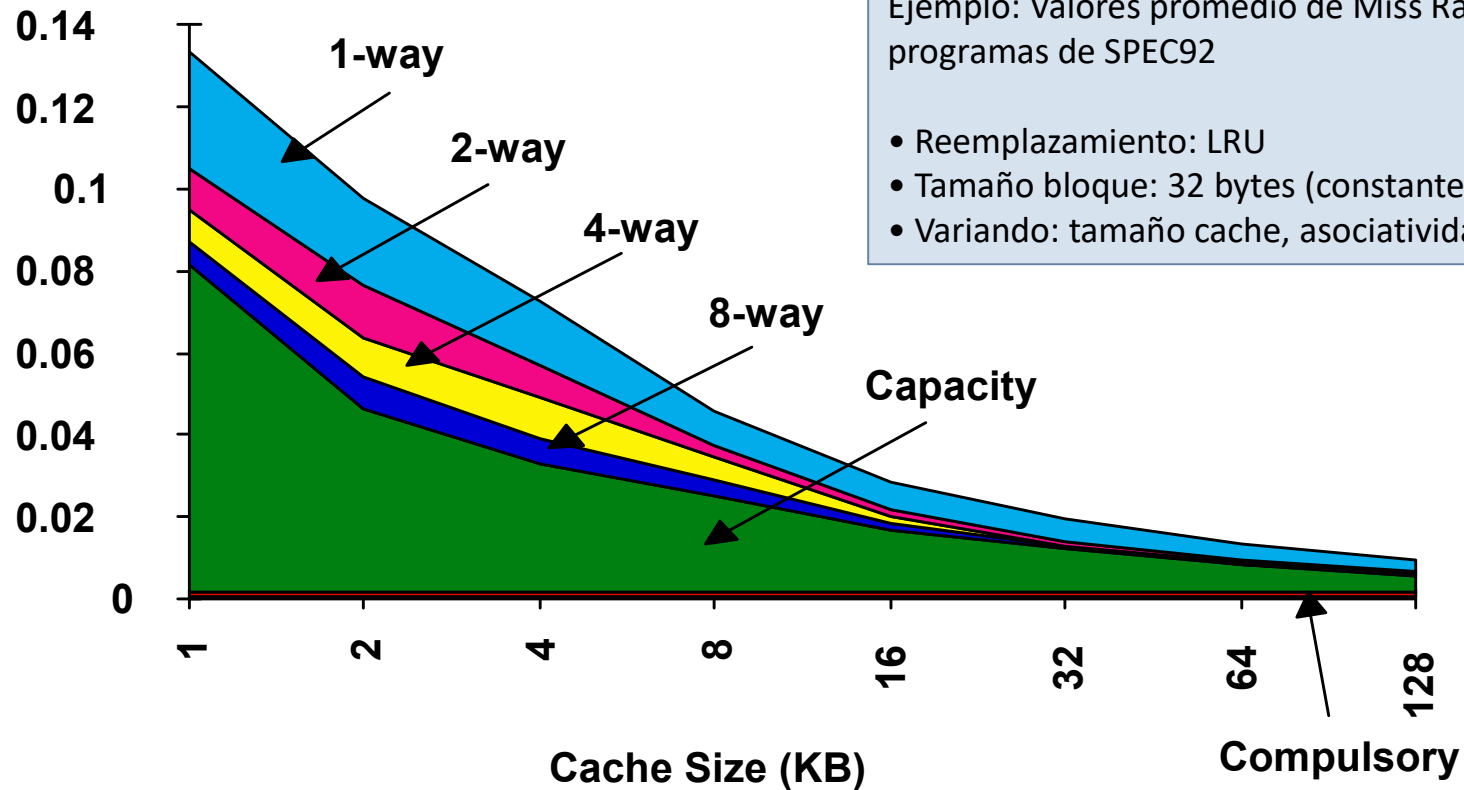
# POLÍTICAS DE EMPLAZAMIENTO

Emplazamiento	Ventajas	Inconvenientes
<b>Directo</b>	Acceso simple y rápido	Alta tasa de fallos cuando varios bloques compiten por el mismo marco
<b>Asociativo</b>	Máximo aprovechamiento de la MC	Una alta asociatividad impacta directamente en el tiempo de acceso a la MC
<b>Asociativo por conjuntos</b>	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 más común: 2	Al aumentar el grado de asociatividad aumenta el tiempo de acceso y el coste hardware



# POLÍTICAS DE EMPLAZAMIENTO

## Miss rate





- ⊙ Introducción: Jerarquía de memoria
- ⊙ **Memoria cache**
  - ⊙ Políticas de emplazamiento
  - ⊙ **Políticas de actualización**
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos





# POLÍTICA DE ACTUALIZACIÓN

- ◎ **¿Qué ocurre cuando se produce una escritura en memoria?**
  - ◎ ¿Se escribe sólo en la cache? ¿Sólo en la memoria principal? ¿En las dos?
- ◎ **Write-through:** Todas las escrituras actualizan la cache y la memoria
  - ◎ Al reemplazar, se puede eliminar la copia de cache: Los datos están ya en la memoria
  - ◎ Bit de control en la cache: Sólo un bit de validez
- ◎ **Write-back:** Todas las escrituras actualizan sólo la cache
  - ◎ Al reemplazar, no se pueden eliminar los datos de la cache: Deben ser escritos primero en la memoria de siguiente nivel
  - ◎ Bits de control: Bit de validez y bit de sucio



## © ¿Qué ocurre cuando se produce un fallo de escritura?

- © **Write allocate** (con asignación en escritura): en un fallo de escritura se lleva el bloque que se va a escribir a la cache
- © **No-write allocate** (sin asignación en escritura): en un fallo de escritura el dato sólo se modifica en la MP (o nivel de memoria siguiente)



# POLÍTICAS DE ACTUALIZACIÓN

- ⊙ Comparación:
  - ⊙ Write-through:
    - La memoria siempre tiene el último valor
    - Control simple
  - ⊙ Write-back:
    - La memoria puede no contener el valor actualizado
    - Mucho menor ancho de banda necesario, escrituras múltiples en bloque
    - Mejor tolerancia a la alta latencia de la memoria



- ⊙ Introducción: Jerarquía de memoria
- ⊙ **Memoria cache**
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ **Políticas de reemplazamiento**
- ⊙ Rendimiento de la memoria cache
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



# POLÍTICAS DE REEMPLAZAMIENTO

- © **Espacio de reemplazamiento:** conjunto de posibles bloques que pueden ser reemplazados por el nuevo bloque
  - **Directo:** el bloque que reside en el marco que el nuevo bloque tiene asignado. Al no existir alternativas no se requieren algoritmos de reemplazamiento
  - **Asociativo:** cualquier bloque que resida en la cache
  - **Asociativo por conjuntos:** cualquier bloque que resida en el conjunto que el nuevo bloque tiene asignado



# POLÍTICAS DE REEMPLAZAMIENTO

- ⊙ Algoritmos (implementados en hardware):
  - ⊙ **Aleatorio**: se escoge un bloque del espacio de reemplazamiento al azar
  - ⊙ **FIFO**: se sustituye el bloque del espacio de reemplazamiento que lleve más tiempo cargado
  - ⊙ **LRU** (least recently used): se sustituye el bloque del espacio de reemplazamiento que lleve más tiempo sin haber sido referenciado
  - ⊙ **LFU** (least frequently used): se sustituye el bloque del espacio de reemplazamiento que haya sido referenciado en menos ocasiones

¿Complejidad hardware de estos algoritmos?



- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ **Rendimiento de la memoria cache**
- ⊙ Optimización de la memoria cache
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



## MC: RENDIMIENTO

### ⊙ Procesador con memoria perfecta (ideal)

$$\text{○ } T_{\text{cpu}} = N \times \text{CPI} \times tc$$

como  $N \times \text{CPI} = N^{\circ} \text{ ciclos CPU} \rightarrow T_{\text{cpu}} = N^{\circ} \text{ ciclos CPU} \times tc$

### ⊙ Procesador con memoria real

$$\text{○ } T_{\text{cpu}} = (N^{\circ} \text{ ciclos CPU} + N^{\circ} \text{ ciclos espera memoria}) \times tc$$

#### ⊙ Cuántos ciclos de espera por la memoria?

$$\text{○ } N^{\circ} \text{ ciclos espera memoria} = N^{\circ} \text{ fallos} \times \text{Miss Penalty}$$

$$\text{○ } N^{\circ} \text{ fallos} = N^{\circ} \text{ referencias a memoria} \times \text{Miss Rate}$$

$$\text{○ } N^{\circ} \text{ referencias a memoria} = N \times \text{AMPI}$$

(AMPI = Media de accesos a memoria por instrucción)

$$\Rightarrow N^{\circ} \text{ ciclos espera memoria} = N \times \text{AMPI} \times \text{Miss Rate} \times \text{Miss Penalty}$$

#### ⊙ Y finalmente

$$\text{○ } T_{\text{cpu}} = [ (N \times \text{CPI}) + (N \times \text{AMPI} \times \text{Miss Rate} \times \text{Miss Penalty}) ] \times tc$$

$$\text{○ } T_{\text{cpu}} = N \times [ \text{CPI} + (\text{AMPI} \times \text{Miss Rate} \times \text{Miss Penalty}) ] \times tc$$

Define el espacio de diseño para la optimización de Mc





## MC: TIPOS DE FALLOS

### ⊙ Iniciales (compulsory)

- Causados por la primera referencia a un bloque que no está en la cache  
→ Hay que llevar primero el bloque a la cache
- Inevitables, incluso con cache infinita
- No depende del tamaño de la Mc. Sí del tamaño de bloque.

### ⊙ Capacidad

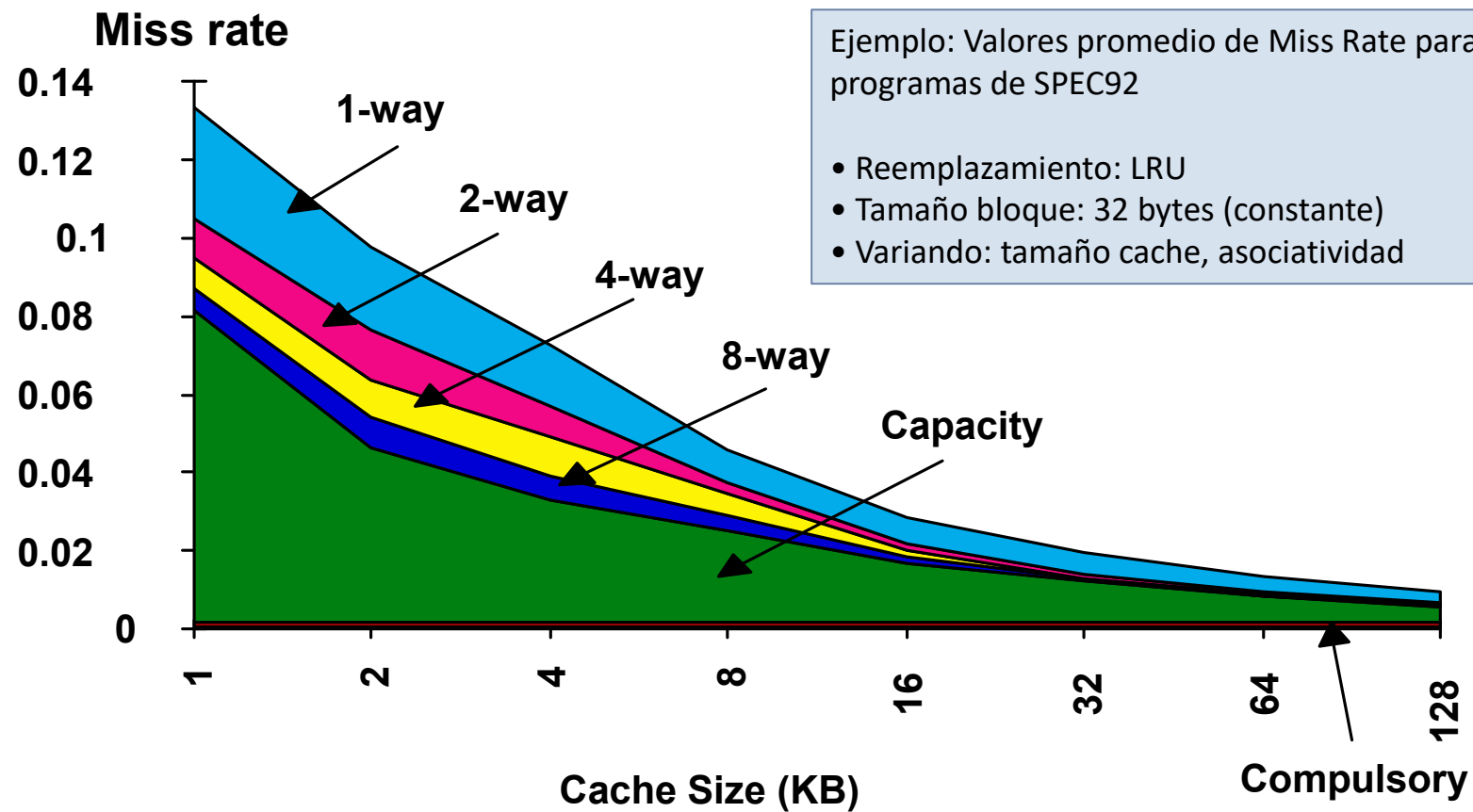
- Si la cache no puede contener todos los bloques necesarios durante la ejecución de un programa, habrá fallos que se producen al recuperar de nuevo un bloque previamente descartado

### ⊙ Conflicto

- Un bloque puede ser descartado y recuperado de nuevo porque hay otro bloque que compite por la misma línea de cache (aunque haya otras líneas libres en la cache)
- No se producen en caches puramente asociativas.



## MC: TIPOS DE FALLOS





- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ **Optimización de la memoria cache**
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



### ⊙ ¿ Como mejorar el rendimiento de la cache?

$$T_{cpu} = N \times [ CPI + (AMPI \times \text{Miss Rate} \times \text{Miss Penalty}) ] \times t_c$$

### ⊙ Estudio de técnicas para:

- ⊙ Reducir la tasa de fallos
- ⊙ Reducir la penalización del fallo
- ⊙ Reducir el tiempo de acierto (hit time)
- ⊙ Aumentar el ancho banda
  - Las dos últimas técnicas inciden sobre  $t_c$



## ESPACIO DE DISEÑO PARA LA MEJORA DEL RENDIMIENTO DE MC

Reducir tasa de fallos	Reducir penalización por fallo	Reducir tiempo de acierto	Aumentar ancho de banda
Tamaño de bloque	Dar prioridad a las lecturas sobre las escrituras	Cache pequeña y sencilla	Cache no bloqueante
Asociatividad	Dar prioridad a la palabra crítica	Predicción de vía	Cache multibanco
Tamaño de Mc	Caches multinivel		Cache segmentada
Algoritmo de reemplazamiento			
Cache de víctimas			
Optimización del código (compilador)			
Prebúsqueda HW			
Prebúsqueda SW			



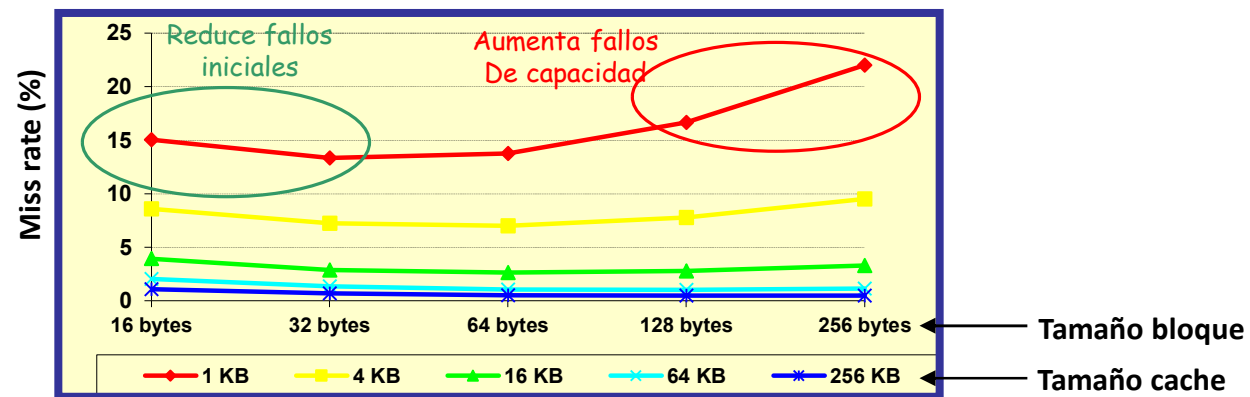
- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ **Optimización de la memoria cache**
  - ⊙ **Reducción de la tasa de fallos de la cache**
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



# REDUCIR LA TASA DE FALLOS

## ⊙ Aumento del tamaño del bloque

- ⊙ Disminución de la **tasa de fallos iniciales** y **captura mejor la localidad espacial**
- ⊙ Aumento de la **tasa de fallos de capacidad y conflicto** (menor  $N^{\circ}$  Bloques => captura peor localidad temporal)
- ⊙ Aumenta la penalización por fallo

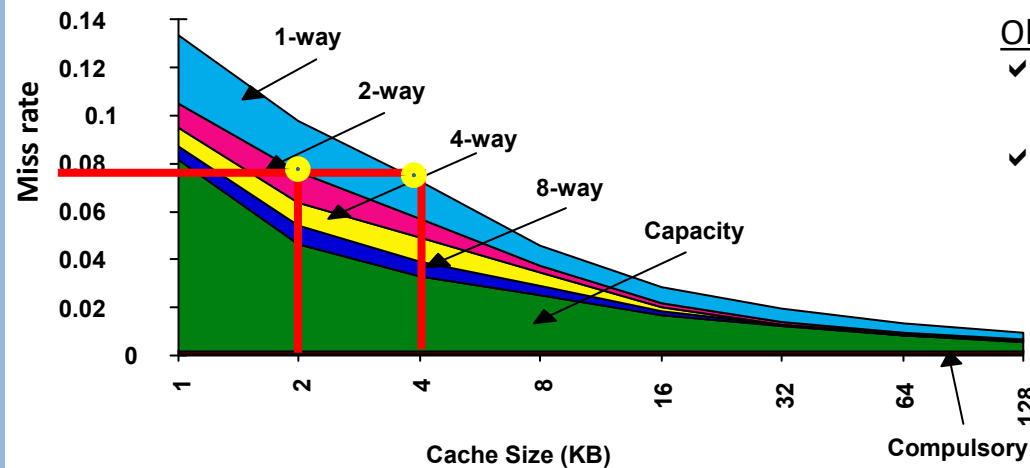




# REDUCIR LA TASA DE FALLOS

## ⊙ Aumento de la asociatividad

- ⊙ Disminución de la **tasa de fallos de conflicto** (más marcos posibles)
- ⊙ Mayor Tiempo de acierto



### Observaciones sobre la tasa de fallos

- ✓ Regla 2:1  
Cache directa= 2-vías de mitad de tamaño
- ✓ 8-vías es igual a totalmente asociativa





## REDUCIR LA TASA DE FALLOS

- ⊙ **Aumento del tamaño de la Mc**
  - ⊙ Obviamente mejora la tasa de fallos (reducción de fallos de capacidad)
  - ⊙ Puede empeorar tiempo de acceso, coste y consumo de energía



# REDUCIR LA TASA DE FALLOS

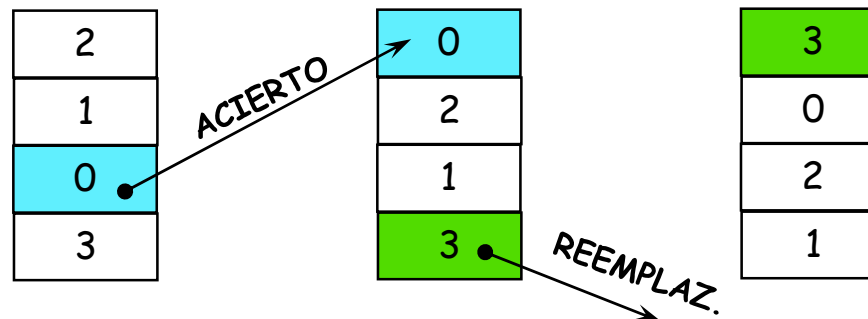
## Selección del algoritmo de reemplazamiento

### Espacio de reemplazamiento:

- **Directo:** trivial
- **Asociativo:** toda la cache
- **Asociativo por conjuntos:** las líneas de un conjunto

### Algoritmos

- **Aleatorio:** El bloque reemplazado se escoge aleatoriamente
- **LRU (*Least Recented Used*):** Se reemplaza el bloque menos recientemente usado. **Gestión:** pila





## REDUCIR LA TASA DE FALLOS

- ◎ **Selección del algoritmo de reemplazamiento (cont.)**
  - ◎ **Técnicas de implementación:** registros de edad, implementación de la pila, etc  
Para un grado de asociatividad mayor que 4, muy costoso en tiempo y almacenamiento (actualización > tcache)  
**LRU aproximado:** Algoritmo LRU en grupos y dentro del grupo
  - ◎ Disminuye la **tasa de fallos de capacidad** (mejora la localidad temporal)

Ejemplo: Fallos de datos por 1000 instrucciones en arquitectura Alpha ejecutando 10 programas SPEC 2000 (tamaño de bloque: 64 bytes)

N° Vías	2		4		8	
Tamaño Mc	LRU	Aleatorio	LRU	Aleatorio	LRU	Aleatorio
16K	114.1	117.3	111.7	115.1	109.0	111.8
64K	103.4	104.3	102.4	102.3	99.7	100.5
256K	92.2	92.1	92.1	92.1	92.1	92.1

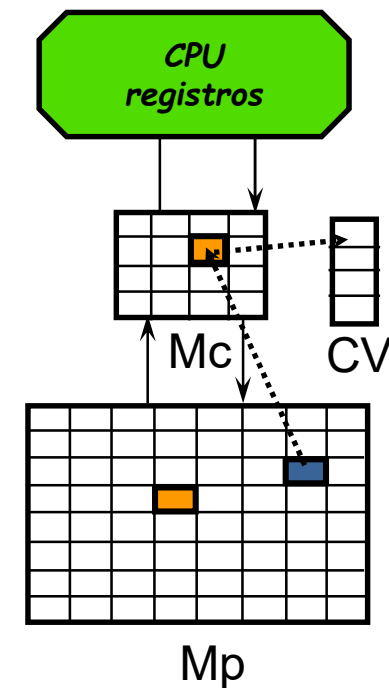
La diferencia LRU-Aleatorio disminuye al aumentar Mc



## REDUCIR LA TASA DE FALLOS

### ◎ Cache de víctimas

- ◎ Objetivo: mantener la sencillez y rapidez de acceso de una MC con emplazamiento directo, pero disminuyendo el impacto de los fallos de conflicto.
- ◎ Es una memoria cache **más pequeña y totalmente asociativa** asociada a la memoria cache
  - Contiene los bloques que han sido **sustituídos más recientemente**
  - En un fallo primero **comprueba** si el bloque se encuentra en la cache de víctimas. En caso afirmativo, el bloque buscado se lleva de la cache de víctimas a la MC.
  - Cuanto menor es la memoria cache más efectiva es la cache víctima





## REDUCIR LA TASA DE FALLOS

### ⊙ **Compilador: Optimización de código**

- ⊙ Todas las optimizaciones las haremos con este ejemplo:
  - DEC Alpha 21064:
    - ⊙ MC de 8 Kbytes
    - ⊙ Emplazamiento directo
    - ⊙ 256 bloques.
    - ⊙ Palabras de 8 bytes
      - Por tanto 1 bloque = 32 bytes = 4 palabras de 8 bytes
      - MC tiene 1024 palabras



# REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

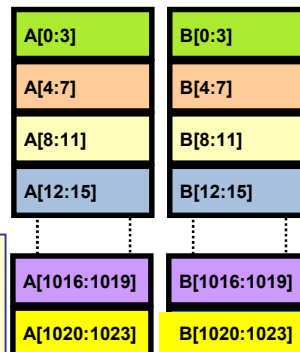
## 1) Fusión de arrays: Mejora la localidad espacial para disminuir los fallos

- Colocar las mismas posiciones de diferentes arrays en posiciones contiguas de memoria

```
double A[1024];  
double B[1024];  
  
for (i = 0; i < 1024; i = i + 1)  
    C = C + (A[i] + B[i]);
```

Todos los accesos son fallos ya que cualquier referencia A[i] se coloca en el mismo Marco de bloque que la B[i]

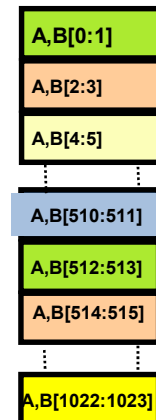
**2x1024 fallos**  
2x256 de inicio  
Resto 2x3x256



```
struct fusion{  
    double A;  
    double B;  
} array[1024];  
  
for (i = 0; i < 1024; i = i + 1)  
    C = C + (array[i].A + array[i].B);
```

**1024/2 fallos**  
2x256 de inicio

**Ganancia: 4**





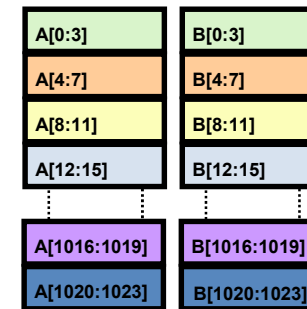
## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

### 2) Alargamiento de arrays: Mejora la localidad espacial para disminuir los fallos

- Impedir que en cada iteración del bucle se compita por el mismo marco de bloque

```
double A[1024];  
double B[1024];  
for (i=0; i < 1024; i=i +1)  
    C = C + (A[i] + B[i]);
```

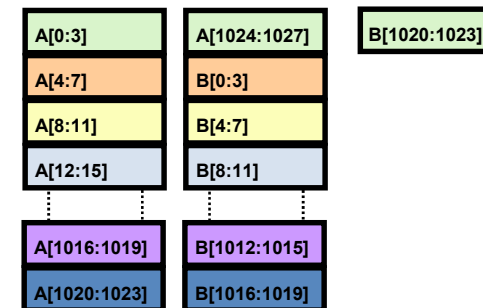
**2x1024 fallos**  
512 de inicio  
Resto 3x512



```
double A[1028];  
double B[1024];  
for (i=0; i < 1024; i=i+1)  
    C = C + (A[i] + B[i]);
```

**1024/2 fallos**  
2x256 de inicio

**Ganancia: 4**





## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

### 3) Intercambio de bucles: Mejora la localidad espacial para disminuir los fallos

- En lenguaje C las matrices se almacenan por filas, luego se debe variar en el bucle interno la columna

```
double A[128][128];  
  
for (j=0; j < 128; j=j+1)  
    for (i=0; i < 128; i=i+1)  
        C = C * A[i][j];
```

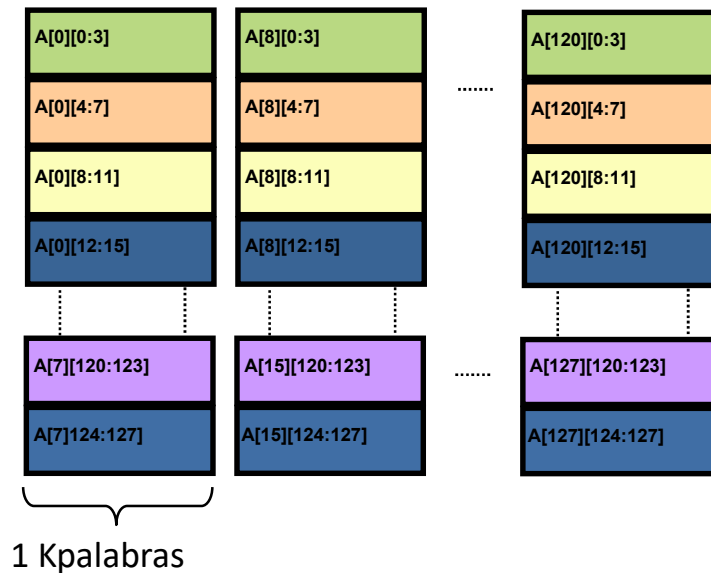
**128x128 fallos**  
16x256 de inicio  
Resto (12288)  
No aprovecha la localidad espacial

```
double A[128][128];  
  
for (i=0; i < 128; i=i+1)  
    for (j=0; j < 128; j=j+1)  
        C = C * A[i][j];
```

**128x128/4 fallos**  
16x256 de inicio

**Ganancia: 4**

A tiene  $2^{14}$  palabras = 16 Kpalabras => es 16 veces mayor que Mc







## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

### 4) Fusión de bucles: Mejora la localidad temporal para disminuir los fallos

- Fusionar los bucles que usen los mismos arrays para usar los datos que se encuentran en cache antes de desecharlos

```
double A[64][64];
for (i=0; i < 64; i=i+1)
    for (j=0; j < 64; j=j+1)
        C = C * A[i][j];
for (i=0; i < 64; i=i+1)
    for (j=0; j < 64; j=j+1)
        D = D + A[i][j];
```

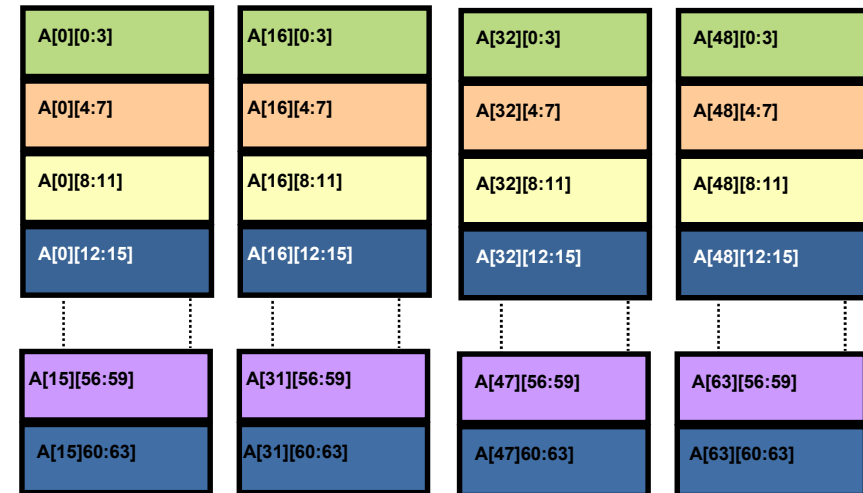
**(64x64/4)x2 fallos**  
4x256 de inicio  
Resto (4x256)

```
double A[64][64];
for (i=0; i < 64; i=i+1)
    for (j=0; j < 64; j=j+1)
    {
        C = C * A[i][j];
        D = D + A[i][j];
    }
```

**64x64/4 fallos**  
4x256 de inicio

**Ganancia: 2**

A tiene  $2^{12}$  palabras = 4 Kpalabras => es  
4 veces mayor que Mc



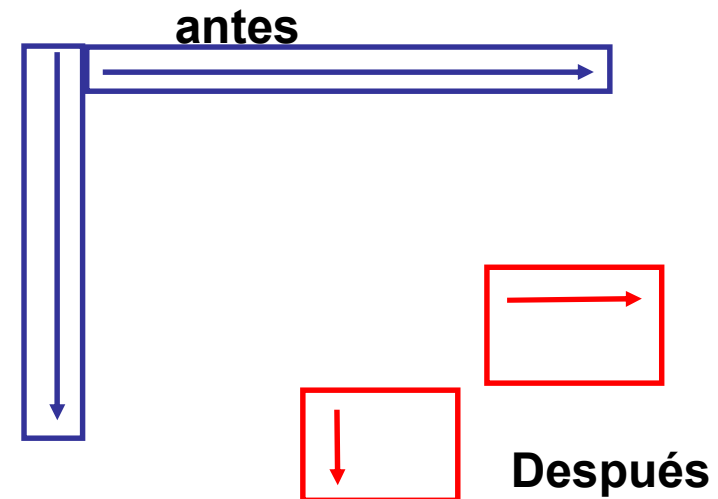


## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

### 5) Cálculo por bloques( Blocking): Mejora la localidad temporal para disminuir los fallos de capacidad

```
/* 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;  
    };
```

- ⊙ Dos bucles internos. Para cada valor de  $i$ :
  - ⊙ Lee todos los  $N \times N$  elementos de  $z$
  - ⊙ Lee  $N$  elementos de 1 fila de  $y$
  - ⊙ Escribe  $N$  elementos de 1 fila de  $x$
- ⊙ Fallos de capacidad dependen de  $N$  y del Tamaño de la cache:
- ⊙ Idea: calcular por submatrices  $B \times B$  que permita el tamaño de la cache





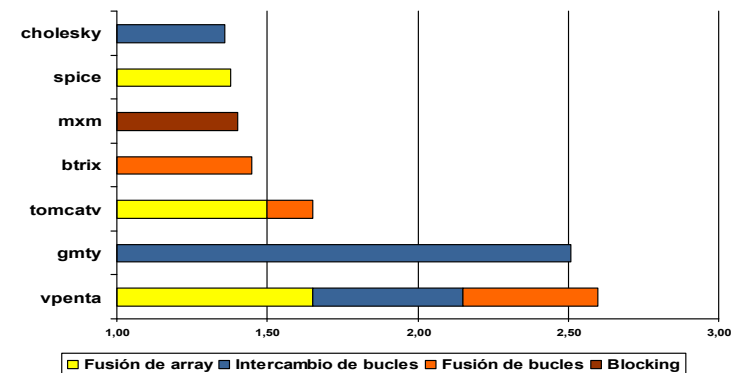
## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

**5) Calculo por bloques( Blocking):** Mejora la localidad temporal para disminuir los fallos de capacidad

```
/* Despues */
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-1,N); j=j+1)
    {r = 0;
     for (k=kk; k < min(kk+B-1,N); k=k+1)
       r = r + y[i][k]*z[k][j];
     x[i][j] = x[i][j]+r;
    };
```

B Factor de bloque (*Blocking Factor*)

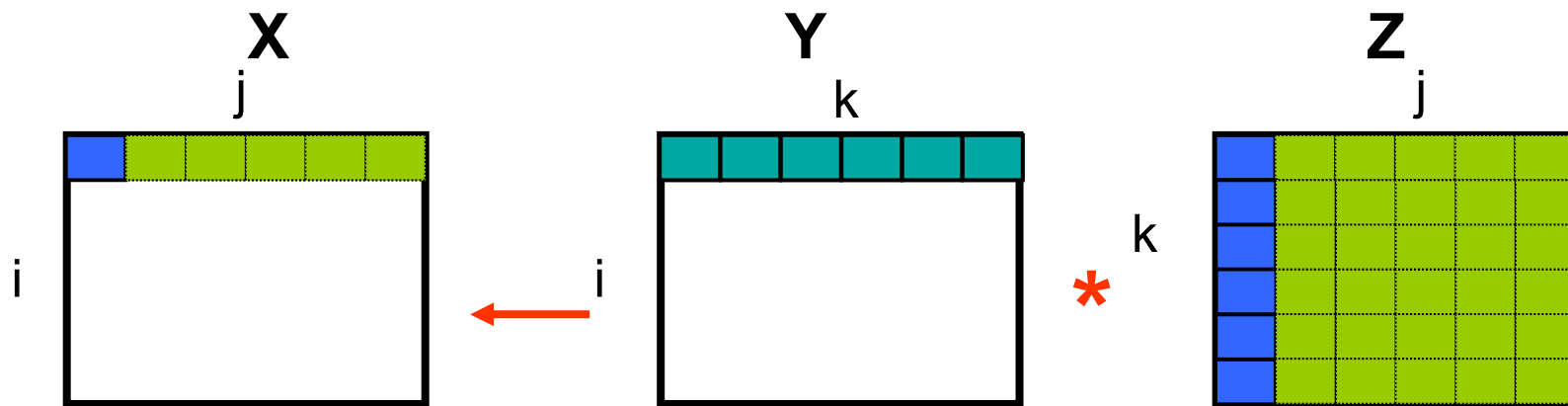
Mejora de rendimiento





## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

- ⊙ Ejemplo: Producto de matrices 6x6 (sin blocking)
  - ⊙ Al procesar la 2ª fila de Y (i=1) se necesita de nuevo la 1ª columna de Z: ¿Está todavía en la cache? Cache insuficiente provoca múltiples fallos sobre los mismos datos



$i = 0, j = 0, k = 0..5$



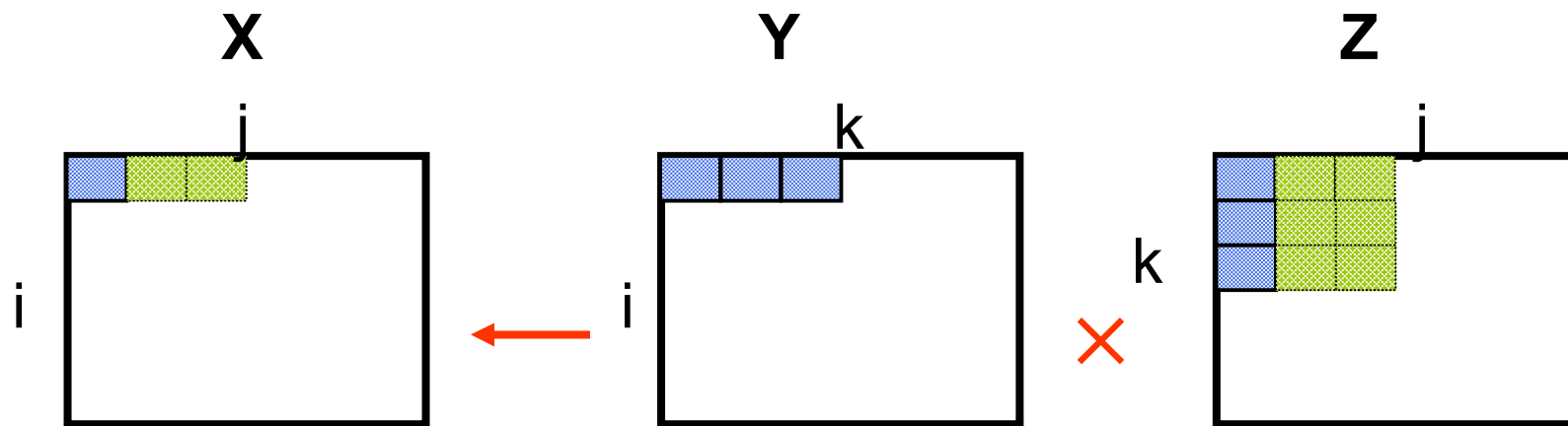
$i = 0, j = 1..5, k = 0..5$

$$X_{ij} = \sum_k Y_{ik} Z_{kj}$$



## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

- ⊙ Ejemplo “blocking”: Con Blocking ( $B=3$ )



$i = 0, j = 0, k = 0..2$



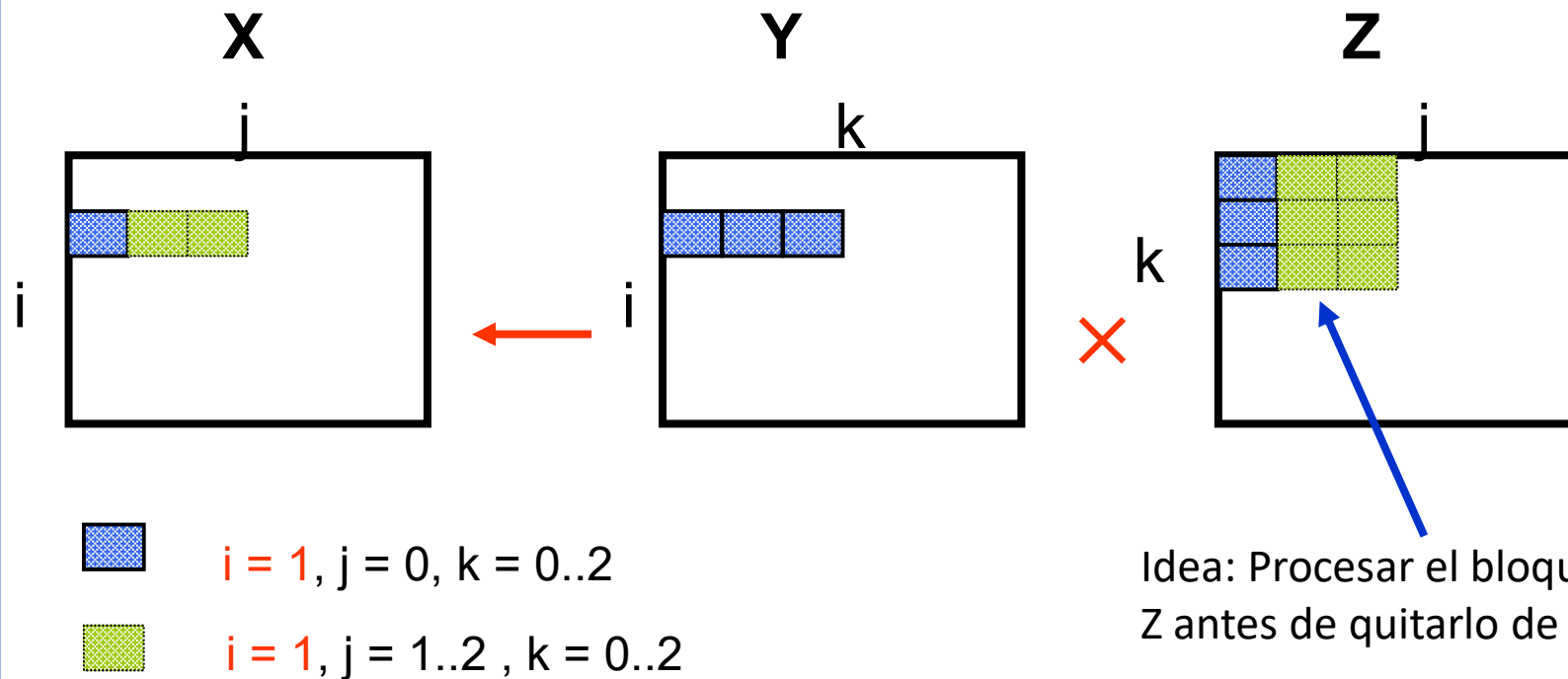
$i = 0, j = 1..2, k = 0..2$

Evidentemente, los elementos de X no están completamente calculados



## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

- ⊙ Ejemplo “blocking”: Con Blocking ( $B=3$ )

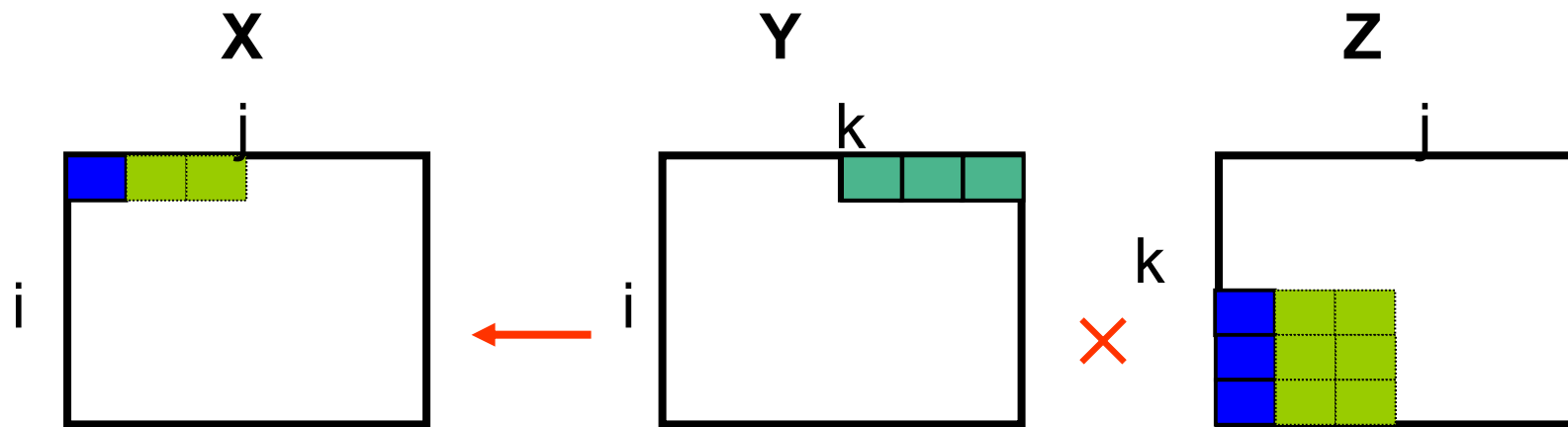




Idea: Procesar el bloque 3x3 de  $Z$  antes de quitarlo de la cache



## REDUCIR LA TASA DE FALLOS: OPTIMIZACIÓN DE CÓDIGO

- © Con Blocking (B=3). Algunos pasos después...



  $i = 0, j = 0, k = 3..5$   
  $i = 0, j = 1..2, k = 3..5$

Y ya empezamos a tener  
elementos de X completamente  
calculados!



# REDUCIR LA TASA DE FALLOS

## ☉ Cache con prebúsqueda

- ☉ Reduce los fallos de Cache anticipando las búsquedas antes de que el procesador demande el dato o la instrucción que provocarían un fallo
  - Se hace una búsqueda en memoria sin que la instrucción o el dato buscado haya sido referenciado por el procesador
    - ☉ Si la información prebuscada se lleva a Mc => **reducción tasa fallos**
    - ☉ Si la información prebuscada se lleva a buffer auxiliar => **reducción penalización**
  - El acceso a memoria se solapa con la ejecución normal de instrucciones en el procesador
  - Existe la posibilidad de que se hagan búsquedas innecesarias
- ☉ **Dos tipos**
  - Prebúsqueda HW
  - Prebúsqueda SW



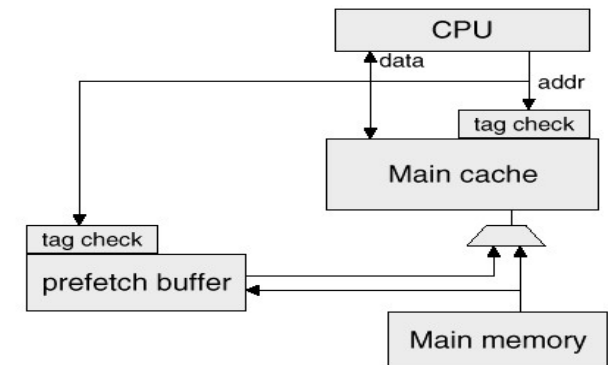


# REDUCIR LA TASA DE FALLOS O LA PENALIZACIÓN POR FALLO

## © Cache con prebúsqueda hardware

- © Prebúsqueda de instrucciones o datos
  - 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

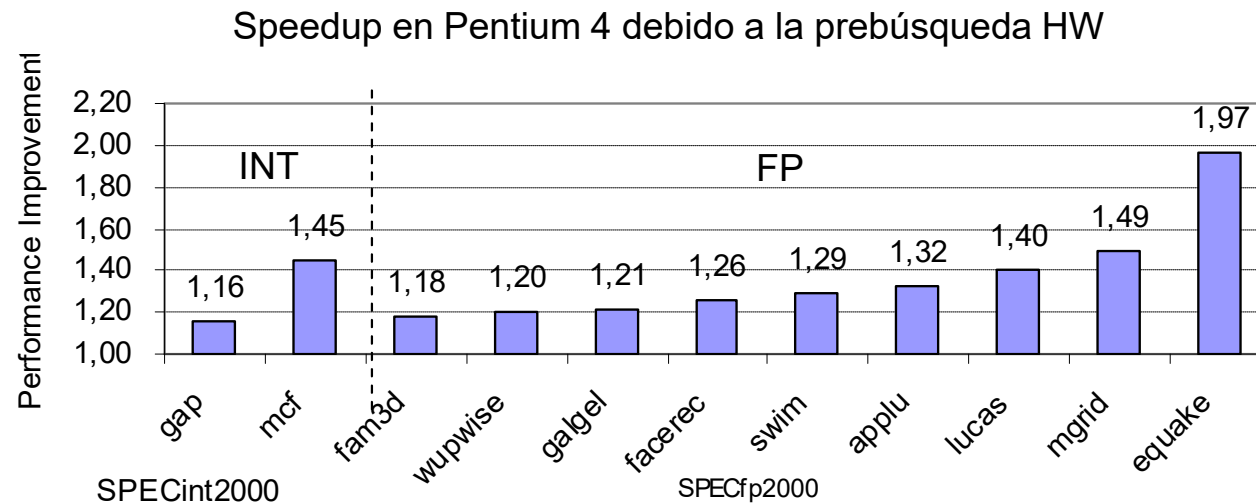
### Implementación (Prebúsqueda de un bloque)





# REDUCIR LA TASA DE FALLOS O LA PENALIZACIÓN POR FALLO

## Cache con prebúsqueda hardware





## REDUCIR LA TASA DE FALLOS O LA PENALIZACIÓN POR FALLO

### ⊙ Cache con prebúsqueda software

- ⊙ Instrucciones especiales de prebúsqueda introducidas por el compilador
- ⊙ La eficiencia depende del compilador y del tipo de programa
- ⊙ Prebúsqueda con destino en cache (MIPS IV, PowerPC, SPARC v. 9)
- ⊙ Instrucciones de prebúsqueda no producen excepciones por fallo de página. Es una forma de especulación.
- ⊙ Funciona bien con bucles y patrones simples de acceso a arrays. Aplicaciones de cálculo
- ⊙ Funciona mal con aplicaciones enteras que presentan un amplio reuso de Cache
- ⊙ Overhead por las nuevas instrucciones. Más búsquedas. Más ocupación de memoria



- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ **Optimización de la memoria cache**
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ **Reducción de la penalización de los fallos de cache**
  - ⊙ Reducción del tiempo de acierto
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



## MEJORA DEL RENDIMIENTO DE MC

Reducir tasa de fallos	Reducir penalización por fallo	Reducir tiempo de acierto	Aumentar ancho de banda
Tamaño de bloque	Dar prioridad a las lecturas sobre las escrituras	Cache pequeña y sencilla	Cache no bloqueante
Asociatividad	Dar prioridad a la palabra crítica	Predicción de vía	Cache multibanco
Tamaño de Mc	Caches multinivel		Cache segmentada
Algoritmo de reemplazamiento			
Cache de víctimas			
Optimización del código (compilador)			
Prebúsqueda HW			
Prebúsqueda SW			



## REDUCIR LA PENALIZACIÓN POR FALLO

### ☉ Dar prioridad a la lecturas sobre las escrituras

- ☉ Un fallo de lectura puede impedir la continuación de la ejecución del programa; un fallo de escritura puede ocultarse
- ☉ **Con escritura directa ( write through)**
  - 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
  - Riesgo: El valor más reciente de una variable, puede estar en buffer y no todavía en MP
  - Ejemplo: cache directa con write-through

```
sw x3, 512(x0);      M[512] <- x3          (bloque 0 de Mc)
lw x1, 1024(x0);     x1 <- M[1024]        (bloque 0 de Mc)
lw x2, 512(x0);      x2 <- M[512]        (fallo lectura: bloque 0 de Mc)
```

(Dependencia LDE en memoria. Con buffer de escrituras ¿tienen x3 y x2 el mismo valor?)
  - En fallo de lectura chequear contenido del buffer de escrituras. Si no hay conflicto, dar prioridad a la lectura y proseguir la ejecución del programa.



## REDUCIR LA PENALIZACIÓN POR FALLO

### ☉ Dar prioridad a la lecturas sobre las escrituras

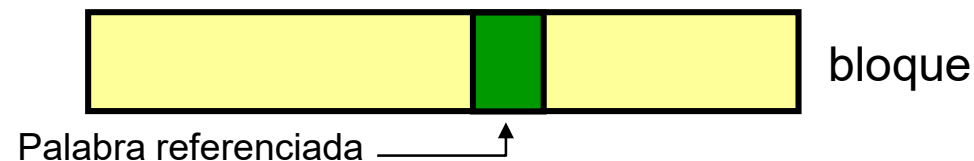
#### ☉ Con post-escritura (write back)

- Se puede aplicar la misma idea, disponiendo de un buffer donde quepa un bloque completo
- Si un fallo de lectura implica reemplazar un bloque sucio => mover bloque sucio a buffer y leer primero bloque en fallo.
- Riesgo: similar al caso anterior



## REDUCIR LA PENALIZACIÓN POR FALLO

- ◎ **Envío directo de la palabra solicitada al procesador**
  - ◎ **Carga anticipada (early restart):** Cuando la palabra solicitada se carga en memoria cache se envía al procesador, sin esperar a la carga del bloque completo
- ◎ **Primero la palabra solicitada (critical word first)**
  - ◎ Primero se lleva al procesador y a memoria cache la palabra solicitada
  - ◎ El resto del bloque se carga en memoria cache en los siguientes ciclos
- ◎ La eficiencia de estas técnicas depende del **tamaño del bloque**. Útil con bloques grandes.
  - ◎ Para bloques pequeños la ganancia es muy pequeña
  - ◎ Problema. Localidad espacial: alta probabilidad de acceder a continuación a la siguiente palabra en secuencia.







## REDUCIR LA PENALIZACIÓN POR FALLO

### ◎ Cache multinivel (L2, L3, ...)

- ◎ Tiempo medio de acceso a memoria (TMAM): Un nivel

- $TMAM = \text{Hit time} + \text{Miss Rate} \times \text{Miss Penalty}$

- ◎ Tiempo medio de acceso a memoria: Dos niveles

- $TMAM = \text{Hit Time L1} + \text{Miss Rate L1} \times \text{Miss Penalty L1}$

- $\text{Miss Penalty L1} = \text{Hit Time L2} + \text{Miss Rate L2} \times \text{Miss Penalty L2} \Rightarrow$

- $\Rightarrow TMAM = \text{Hit Time L1} + \text{Miss Rate L1} \times [\text{Hit Time L2} + (\text{Miss Rate L2} \times \text{Miss Penalty L2})]$



### ◎ Cache multinivel (L1, L2, L3, ...)

#### ◎ Definiciones:

- **Tasa de fallos local en una cache (Lx):** fallos en cache Lx dividido por el número total de accesos a la cache Lx
- **Tasa de fallos global en una cache (Lx):** fallos en cache Lx dividido por el número total de accesos a memoria generados por el procesador
  - Tasa de fallos global en L1 = Tasa de fallos local en L1
  - Tasa de fallos global en L2 ≠ Tasa de fallos local en L2
- La tasa de fallos global es lo importante
  - L1: Afecta directamente al procesador => Acceder a un dato en el ciclo del procesador
  - L2: Afecta a la penalización de L1 => Reducción del tiempo medio de acceso



- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ **Optimización de la memoria cache**
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ **Reducción del tiempo de acierto**
  - ⊙ Aumento del ancho de banda
- ⊙ Ejemplos



## MEJORA DEL RENDIMIENTO DE MC

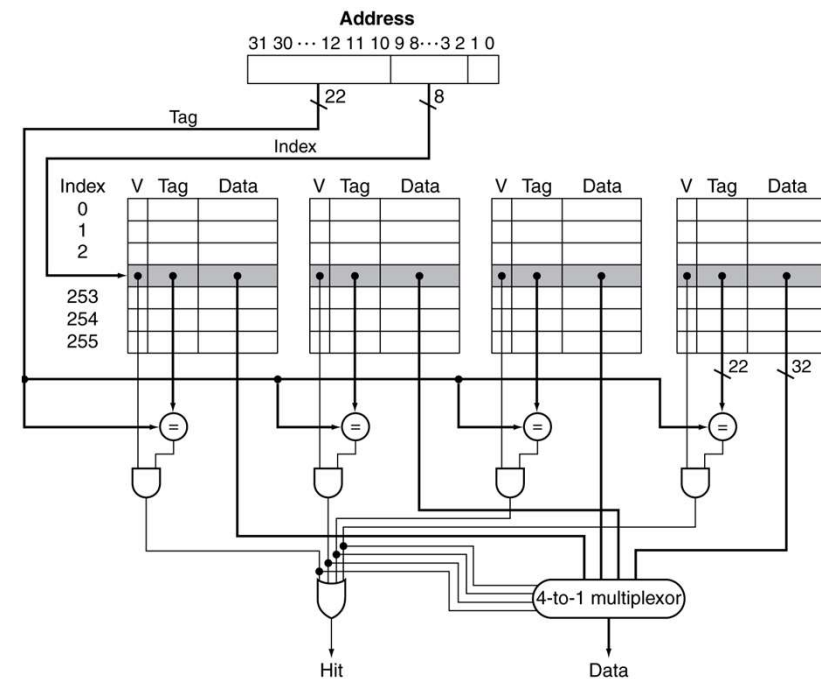
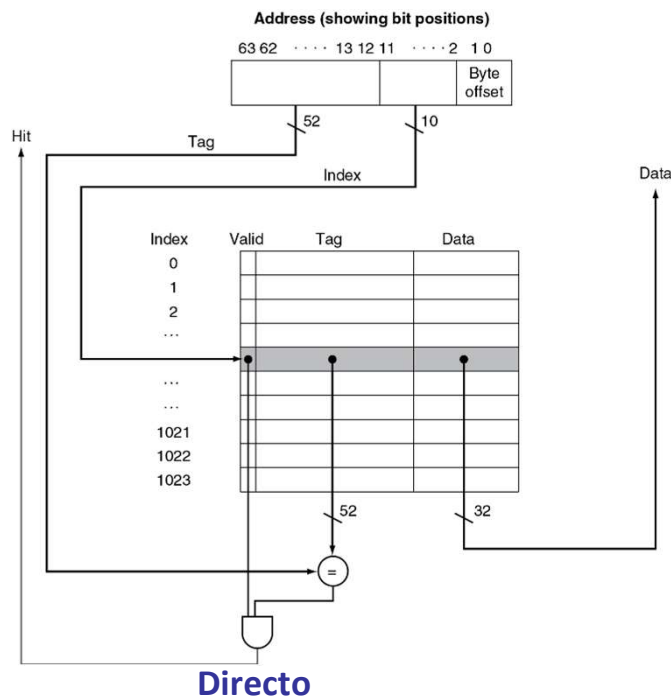
Reducir tasa de fallos	Reducir penalización por fallo	Reducir tiempo de acierto	Aumentar ancho de banda
Tamaño de bloque	Dar prioridad a las lecturas sobre las escrituras	Cache pequeña y sencilla	Cache no bloqueante
Asociatividad	Dar prioridad a la palabra crítica	Predicción de vía	Cache multibanco
Tamaño de Mc	Caches multinivel		Cache segmentada
Algoritmo de reemplazamiento			
Cache de víctimas			
Optimización del código (compilador)			
Prebúsqueda HW			
Prebúsqueda SW			



# REDUCIR EL TIEMPO DE ACIERTO

## © Caches simples y pequeñas

- © El acceso al directorio y la comparación de tags consume tiempo
- © Ejemplo: Comparación de acceso a un dato en cache directa y en cache asociativa por conjuntos con 4 vías



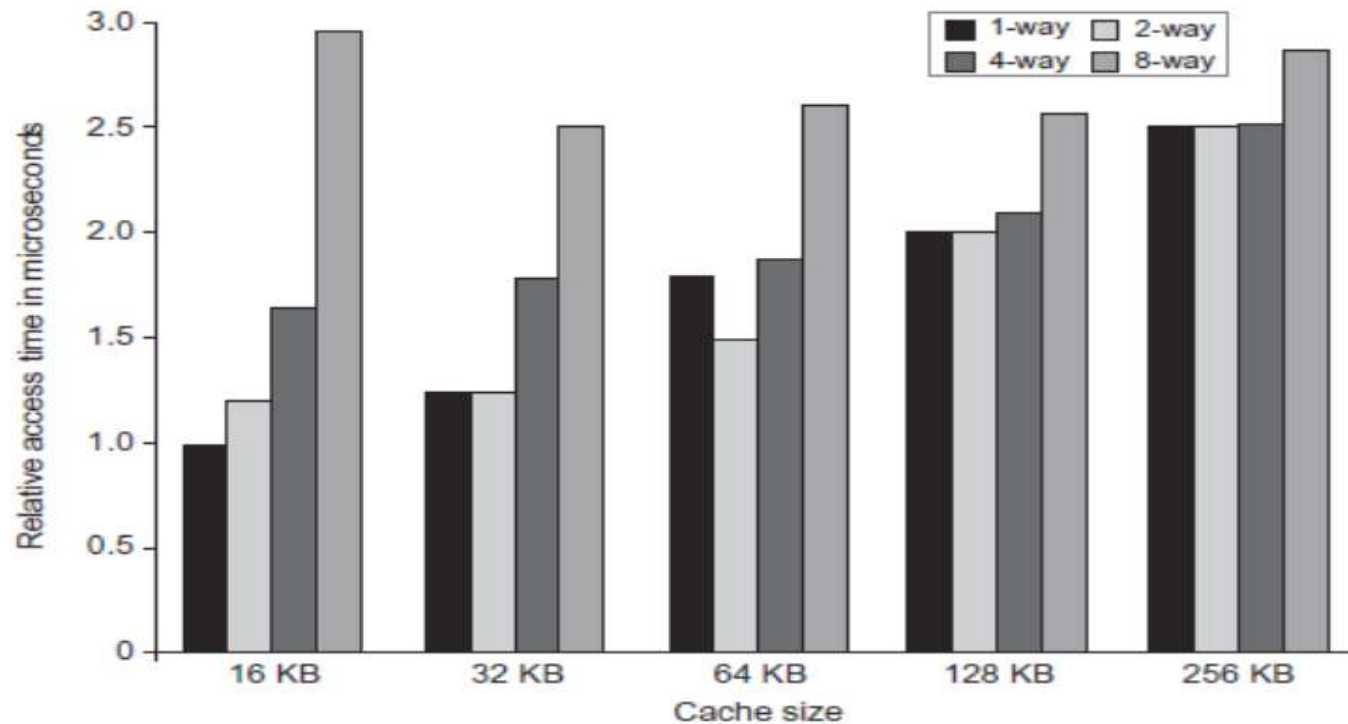


### ◎ Caches simples y pequeñas

- ◎ Una cache pequeña se puede integrar junto al procesador
  - evitando la penalización en tiempo del acceso al exterior
- ◎ Ejemplo: tres generaciones de 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 etiquetas 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 etiquetas
- ◎ Recordar: impacto del tamaño de la cache y la asociatividad sobre el tiempo de acceso



## MEMORIA CACHE: TAMAÑO Y ASOCIATIVIDAD



Tiempo de acceso vs. tamaño y asociatividad (SRAM)



## REDUCIR EL TIEMPO DE ACIERTO

### ◎ Predicción de vía

- ◎ Permite combinar el rápido tiempo de acierto de una cache directa con la menor tasa de fallos de conflicto de una cache asociativa por conjuntos
- ◎ Cada bloque de la cache contiene bits de predicción que indican cuál será la vía más probable del siguiente acceso
- ◎ El multiplexor selecciona la vía predicha antes de completar la comparación de tags
- ◎ En caso de fallo de la predicción, completar la comparación de tags en todas las líneas del conjunto seleccionado

Hit Time (acierto pred)



- ◎ Se han alcanzado tasas de éxito en la predicción en torno al 90% con una cache asociativa por conjuntos con 2 vías
- ◎ Problema: diferentes tiempos en caso de acierto
- ◎ Ejemplo: Se utiliza en R10000, Pentium 4, ARM Cortex-A8,..





- ⊙ Introducción: Jerarquía de memoria
- ⊙ Memoria cache
  - ⊙ Políticas de emplazamiento
  - ⊙ Políticas de actualización
  - ⊙ Políticas de reemplazamiento
- ⊙ Rendimiento de la memoria cache
- ⊙ **Optimización de la memoria cache**
  - ⊙ Reducción de la tasa de fallos de la cache
  - ⊙ Reducción de la penalización de los fallos de cache
  - ⊙ Reducción del tiempo de acierto
  - ⊙ **Aumento del ancho de banda**
- ⊙ Ejemplos



## MEJORA DEL RENDIMIENTO DE MC

Reducir tasa de fallos	Reducir penalización por fallo	Reducir tiempo de acierto	Aumentar ancho de banda
Tamaño de bloque	Dar prioridad a las lecturas sobre las escrituras	Cache pequeña y sencilla	Cache no bloqueante
Asociatividad	Dar prioridad a la palabra crítica	Predicción de vía	Cache multibanco
Tamaño de Mc	Caches multinivel		Cache segmentada
Algoritmo de reemplazamiento			
Cache de víctimas			
Optimización del código (compilador)			
Prebúsqueda HW			
Prebúsqueda SW			



### ◎ Cache sin bloqueo ( non-blocking, lockup-free )

#### ◎ Idea:

- Ocultar la latencia de un fallo de cache solapándolo con otras instrucciones independientes

ADD     x5, x6, x6

.....

.....

LW       x1, dir

.....

.....

ADD     x4,x4,x1



NO BLOQUEAR: Cache que no bloquean  
(Se siguen ejecutando instrucciones después del  
LW. El ADD no se ejecuta hasta que x1 está  
disponible)



## AUMENTAR EL ANCHO DE BANDA

### ⊙ **Cache sin bloqueo ( non-blocking, lockup-free)**

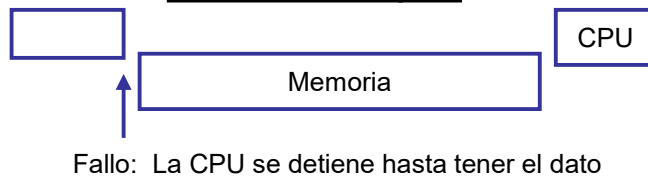
- ⊙ Permite que la ejecución siga aunque se produzca un fallo mientras no se necesite el dato. (Se aplica a la cache de datos).
- ⊙ Un fallo sin servir (hit under 1 miss). Sigue ejecutando y proporcionando datos que están en cache
  - HP7100, Alpha 21064
- ⊙ Múltiples fallos sin servir (hit under multiple misses)
  - R12000 (4) , Alpha21264 (8), HP8500 (10), Pentium-III y 4 ( 4), Sandy-Bridge (10)
- ⊙ Los beneficios dependen de la planificación de instrucciones
- ⊙ Requiere interfaz de memoria más complejo ( múltiples bancos )



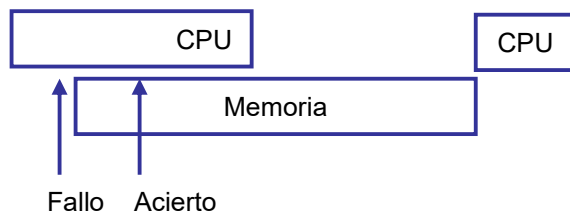
# AUMENTAR EL ANCHO DE BANDA

## ☉ Cache sin bloqueo ( non-blocking, lockup-free)

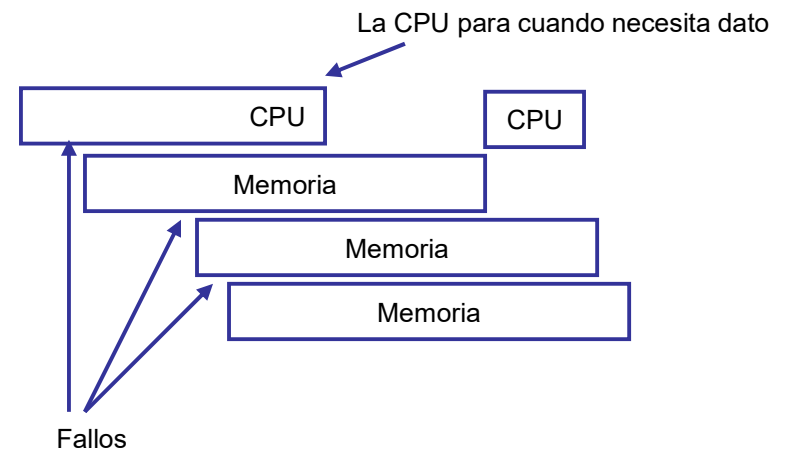
### Cache con bloqueo



### Cache sin bloqueo: un fallo sin servir



### Cache sin bloqueo: varios fallos sin servir





## AUMENTAR EL ANCHO DE BANDA

### ☉ Cache multibanco

- ☉ Dividir la cache en bancos independientes que puedan soportar accesos simultáneos.
  - Ejemplo: L2 de SUN T1 (Niágara) tiene 4 bancos, L2 de AMD Opteron tiene 2 bancos
- ☉ La organización en bancos funciona bien cuando los accesos se dispersan de forma natural entre los diferentes bancos
- ☉ Direcciones consecutivas están en bancos consecutivos
- ☉ Ejemplo: Ubicación de bloques en una cache con 4 bancos con entrelazamiento de orden bajo

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

© 2007 Elsevier, Inc. All rights reserved.

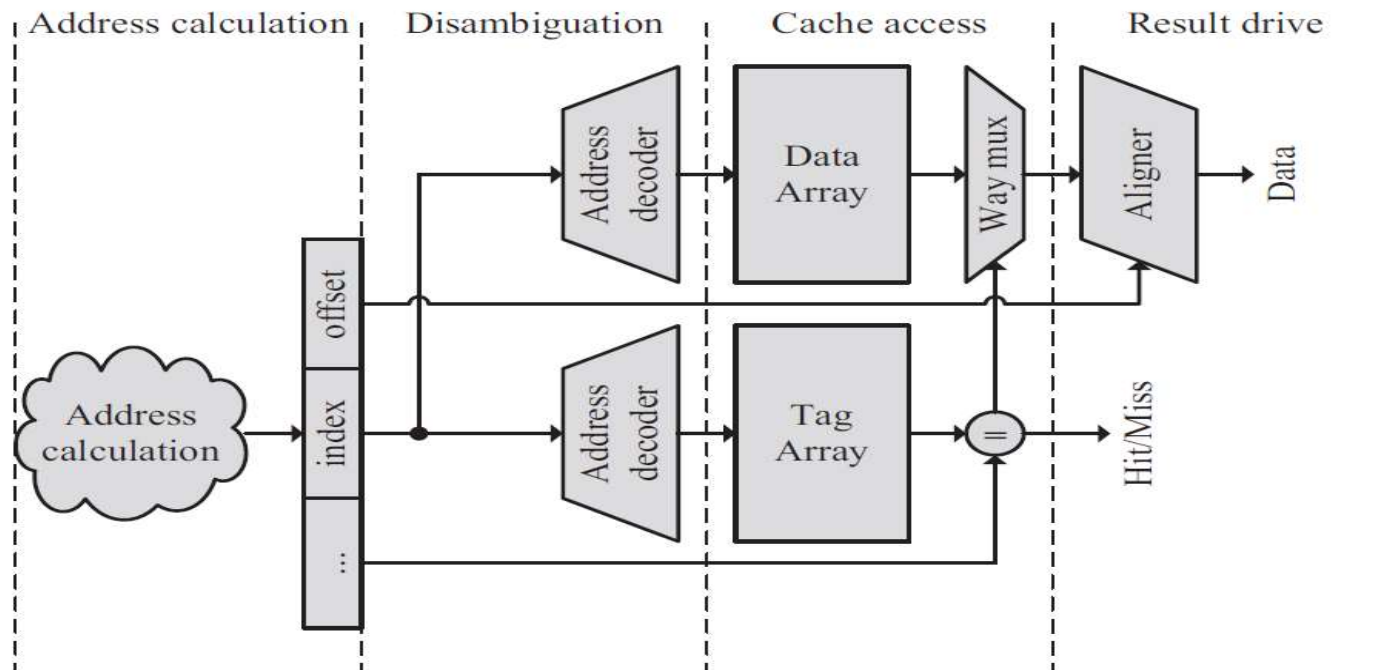


### ◎ Cache segmentada

- ◎ Segmentar los accesos a la cache permite aumentar el ancho de banda.
- ◎ Problema: incremento de los ciclos de latencia. Más ciclos de reloj entre el lanzamiento de un LDR y el uso de los datos que el LDR proporciona
- ◎ Ejemplos: N° de etapas del acceso a la cache en diferentes procesadores
  - Pentium 1 etapa
  - De Pentium Pro a Pentium III 2 etapas
  - Pentium 4 y Core i7 4 etapas



## AUMENTAR EL ANCHO DE BANDA



Processor Microarchitecture. An implementation Perspective. A. González et al. Morgan&Claypool Publ. 2011.





## CACHES RESUMEN (I)

Técnica	Tasa fallos	Penal fallo	Tiempo acierto	Ancho banda	Coste HW / Complejidad	Comentario
Aumento tamaño de bloque	+	-			0	Trivial. L2 de Pentium 4 usa 128 bytes
Aumento asociatividad	+		-		1	Ampliamente usado
Aumento tamaño de Mc	+		-		1	Ampliamente usado, especialmente en L2
Mejora algoritmo reemplazamiento	+		-		1	LRU (o pseudo) bastante usado
Cache de víctimas	+	-			1	Bastante sencillo
Optimización del compilador	+				0	El software presenta oportunidades de mejora. Algunos computadores tienen opciones de optimización
Prebúsqueda HW	+	+			2 instr., 3 data	Muchos procesadores prebuscan instrucciones. AMD Opteron y Pentium 4 prebuscan datos.
Prebúsqueda SW	+	+			3	Necesita cache no bloqueante. En muchas CPUs.



## CACHES RESUMEN (II)

Técnica	Tasa fallos	Penal fallo	Tiempo acierto	Ancho banda	Coste HW / Complejidad	Comentario
Prioridad a las lecturas		+			1	Ampliamente usado
Prioridad a la palabra crítica		+			2	Ampliamente usado
Fusión de buffers de escritura		+			1	Ampliamente usado con write through
Cache multinivel		+			2	Ampliamente usado. Más complejo si tamaño de bloque en L1 y L2 distintos.
Cache pequeña y sencilla	–		+		0	Trivial; ampliamente usado.
Predicción de vía			+		1	Usado en Pentium 4
Cache no bloqueante		+		+	3	Ampliamente usado
Cache multibanco				+	1	En L2 de Opteron y Niagara
Cache segmentada			–	+	1	Ampliamente usado



# EJEMPLOS: ARM CORTEX-A8 E INTEL CORE I7 920

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles