Memorias Cache

Estructura de Computadores
Semana 15

Bibliografía:

[BRY16] Cap.6 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/C.1 BRY com

Transparencias del libro CS:APP, Cap.6
Introduction to Computer Systems: a Programmer's Perspective
Autores: Randal E. Bryant y David R. O'Hallaron
http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Guía de trabajo autónomo (4h/s)

- **Lectura:** del Cap.6 CS:APP (Bryant/O'Hallaron)
  - Cache Memories (incl. ***fully associative caches***)
    - § 6.4 pp.650-669
  - Writing Cache-Friendly Code
    - § 6.5 pp.669-674
  - Impact of Caches on Program Performance
    - § 6.6 pp.675-684
- **Ejercicios:** del Cap.6 CS:APP (Bryant/O'Hallaron)
  - Probl. 6.9 § 6.4.1, p.652
  - Probl. 6.10 – 6.11 § 6.4.2, p.660
  - Probl. 6.12 – 6.16 § 6.4.4, pp.664-666
  - Probl. 6.17 – 6.20 § 6.5, pp.672-675
  - Probl. 6.21 § 6.6, p.679

Bibliografía:

[BRY16] Cap.6 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

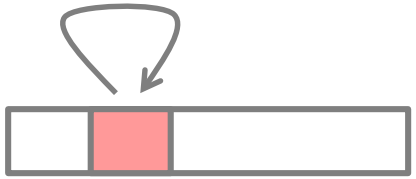
Signatura ESIIT/C.1 BRY com

Memoria II: Cache

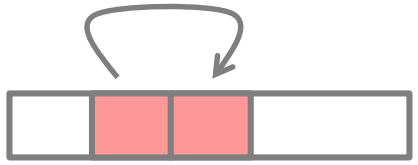
- **Organización y Funcionamiento de la memoria cache**
- **Impacto de la cache en el rendimiento**
  - Modelo de evaluación
  - La montaña de memoria
- **Programación de código aprovechando la cache**

Localidad (Recordatorio)

- **Principio de localidad:** Los programas tienden a usar datos e instrucciones con direcciones iguales o cercanas a las que han usado recientemente

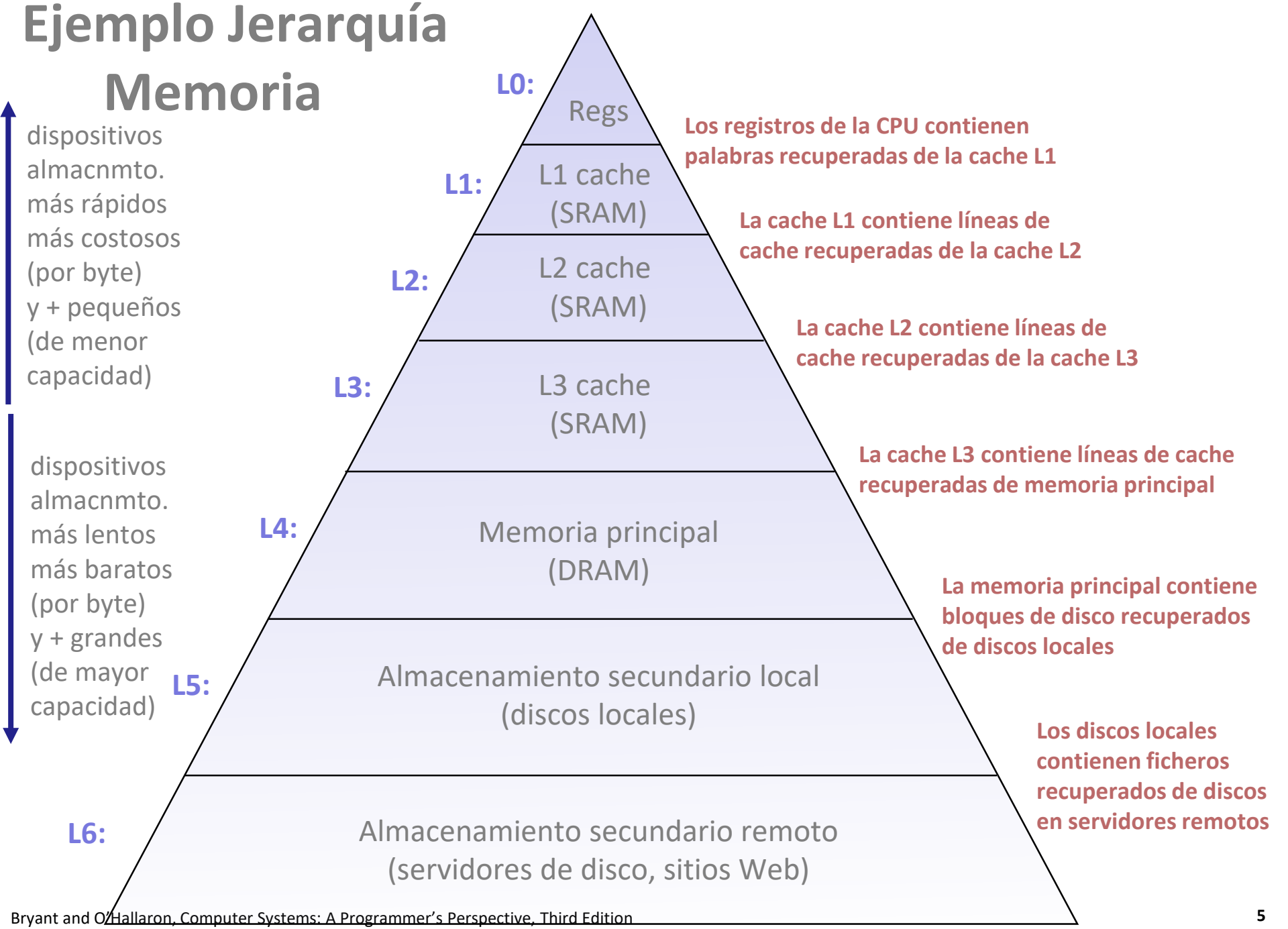


- **Localidad temporal:**
  - Elementos referenciados recientemente probablemente serán referenciados de nuevo en un futuro próximo

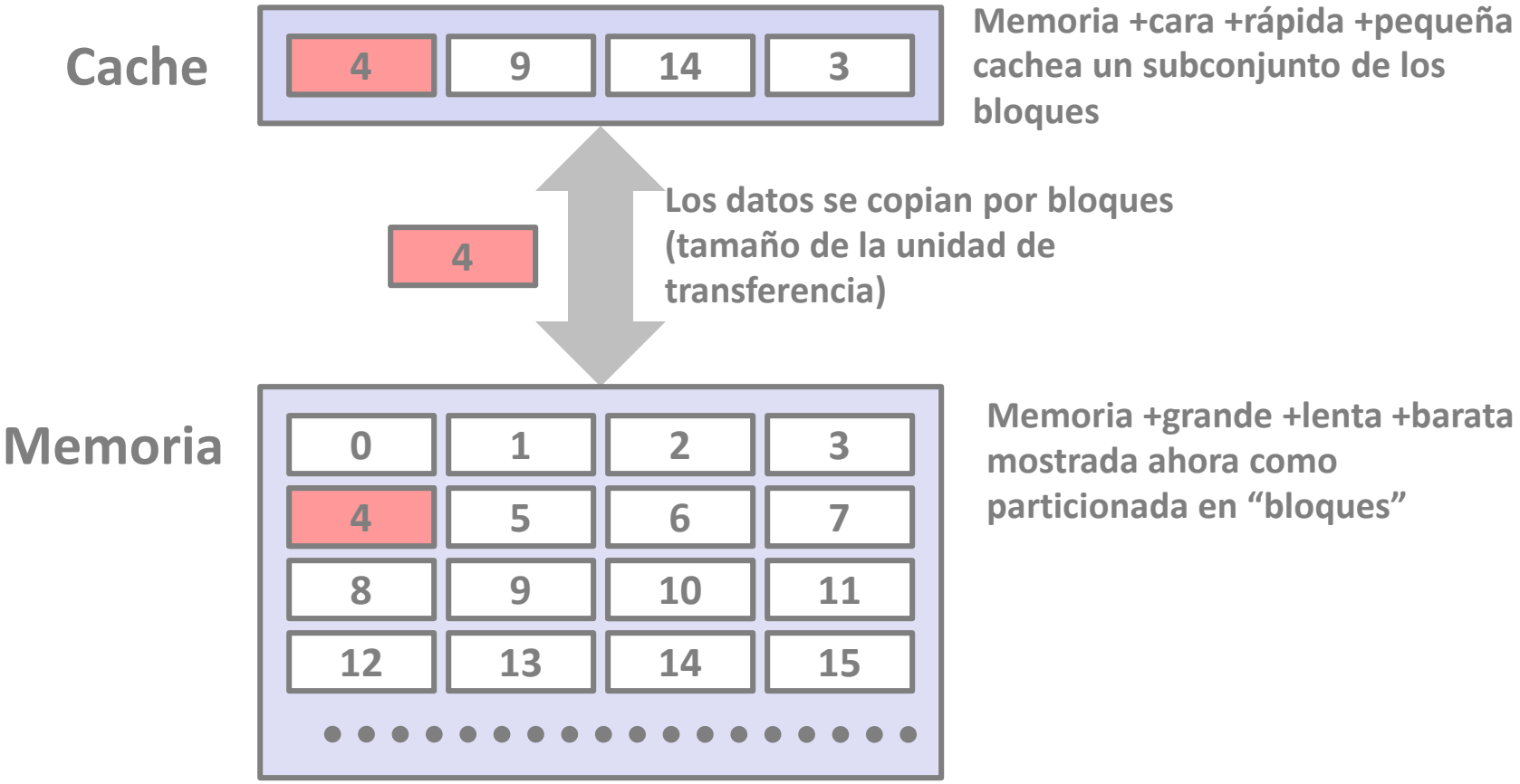


- **Localidad espacial:**
  - Elementos con direcciones cercanas tienden a ser referenciados muy juntos en el tiempo

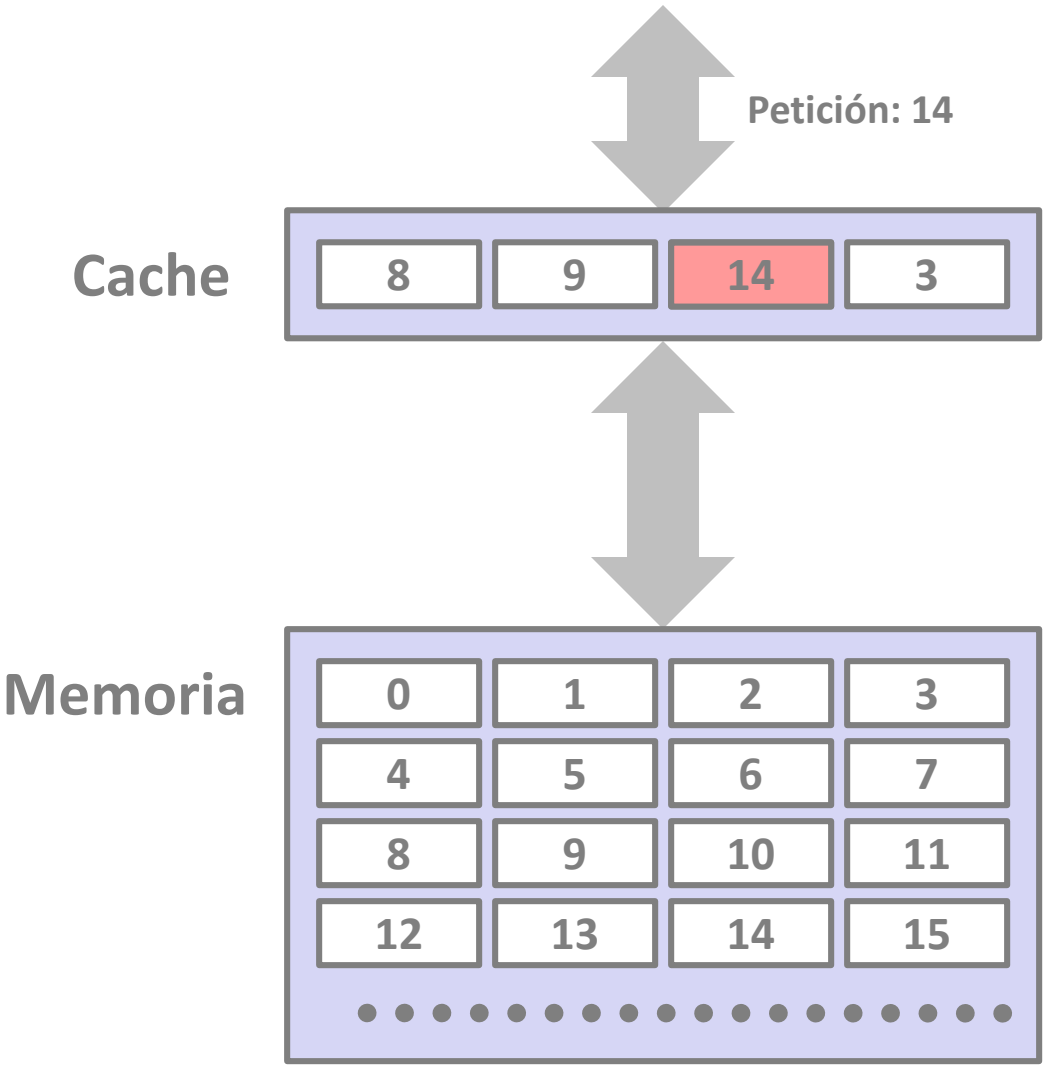
Ejemplo Jerarquía Memoria



Conceptos Generales de Cache



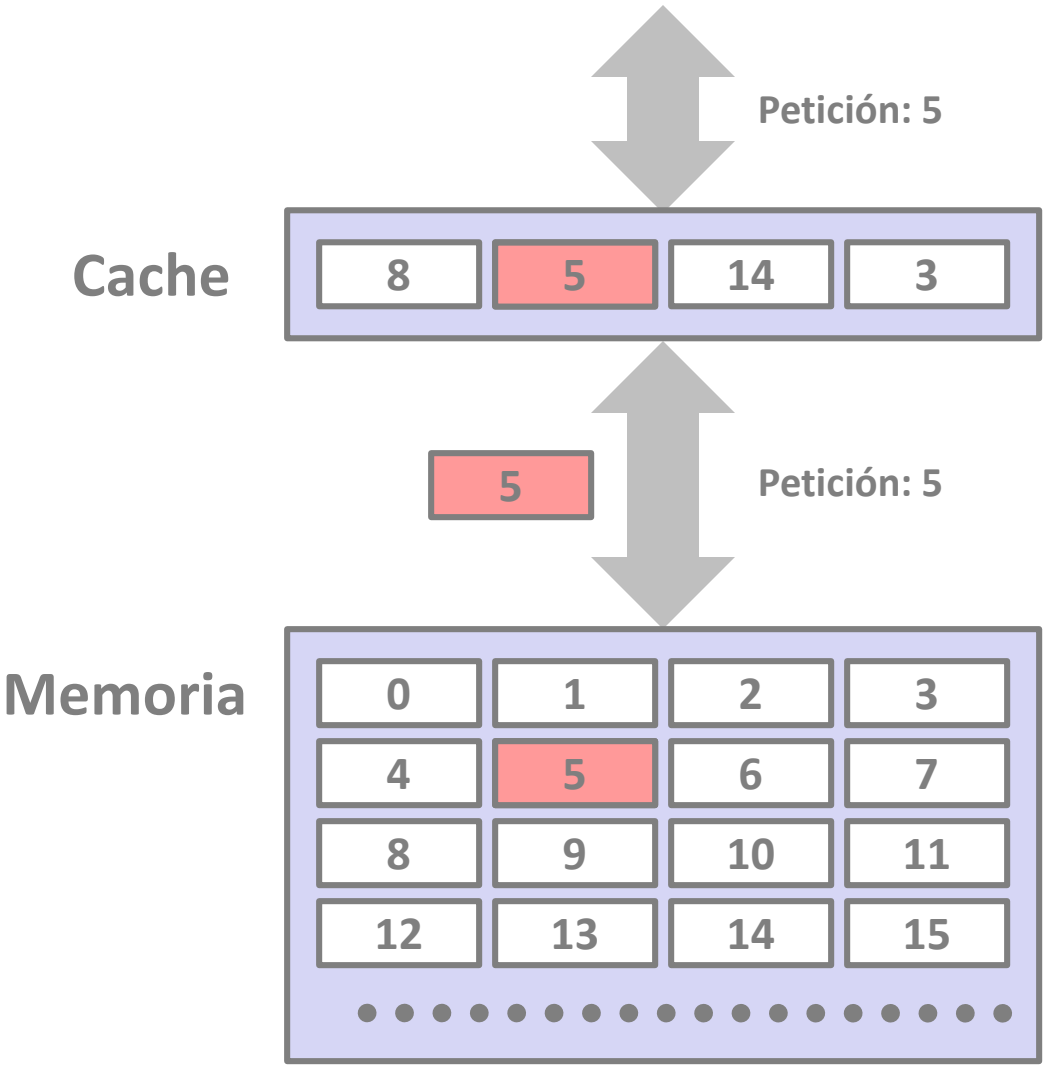
Conceptos Generales de Cache: Acierto†



Se necesitan datos del bloque b

El bloque b está en cache: ¡Acierto!

Conceptos Generales de Cache: Fallo†



Se necesitan datos del bloque b

El bloque b no está en cache: ¡Fallo!

El bloque b se capta de memoria

El bloque b se almacena en cache

- Política de colocación†: determina dónde va b
- Política de reemplazo†: determina qué bloque es desalojado† (víctima)

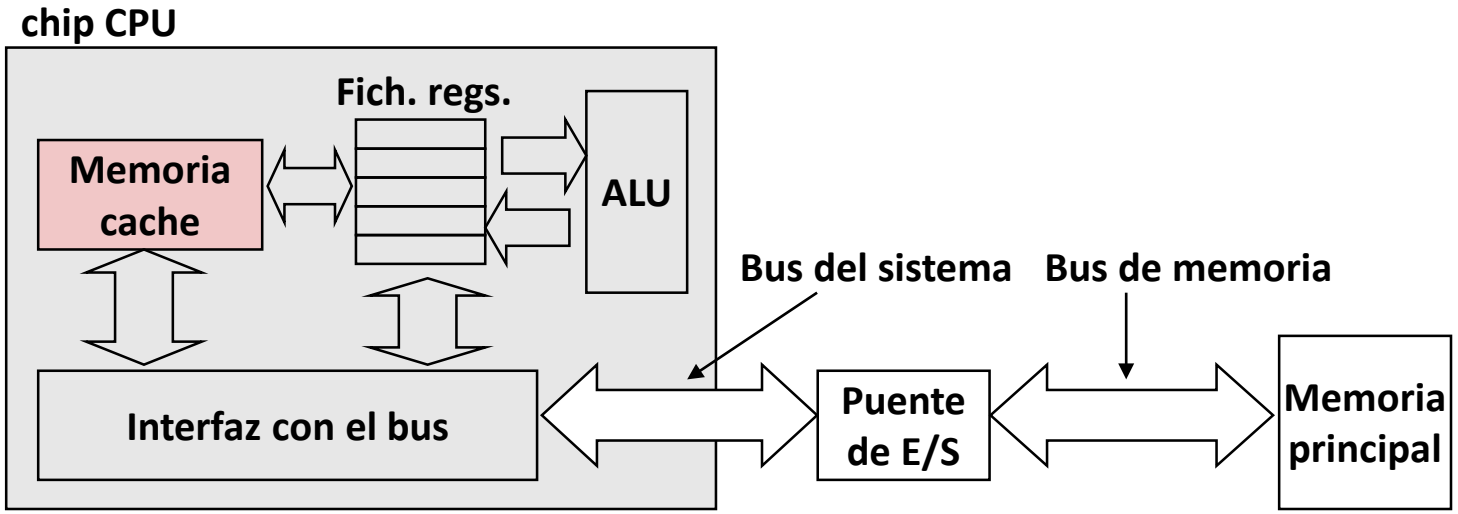
† cache miss [re]placement policy, evicted/victim block

Conceptos Generales de Cache: 3 Tipos de Fallo de Cache (Recordatorio)

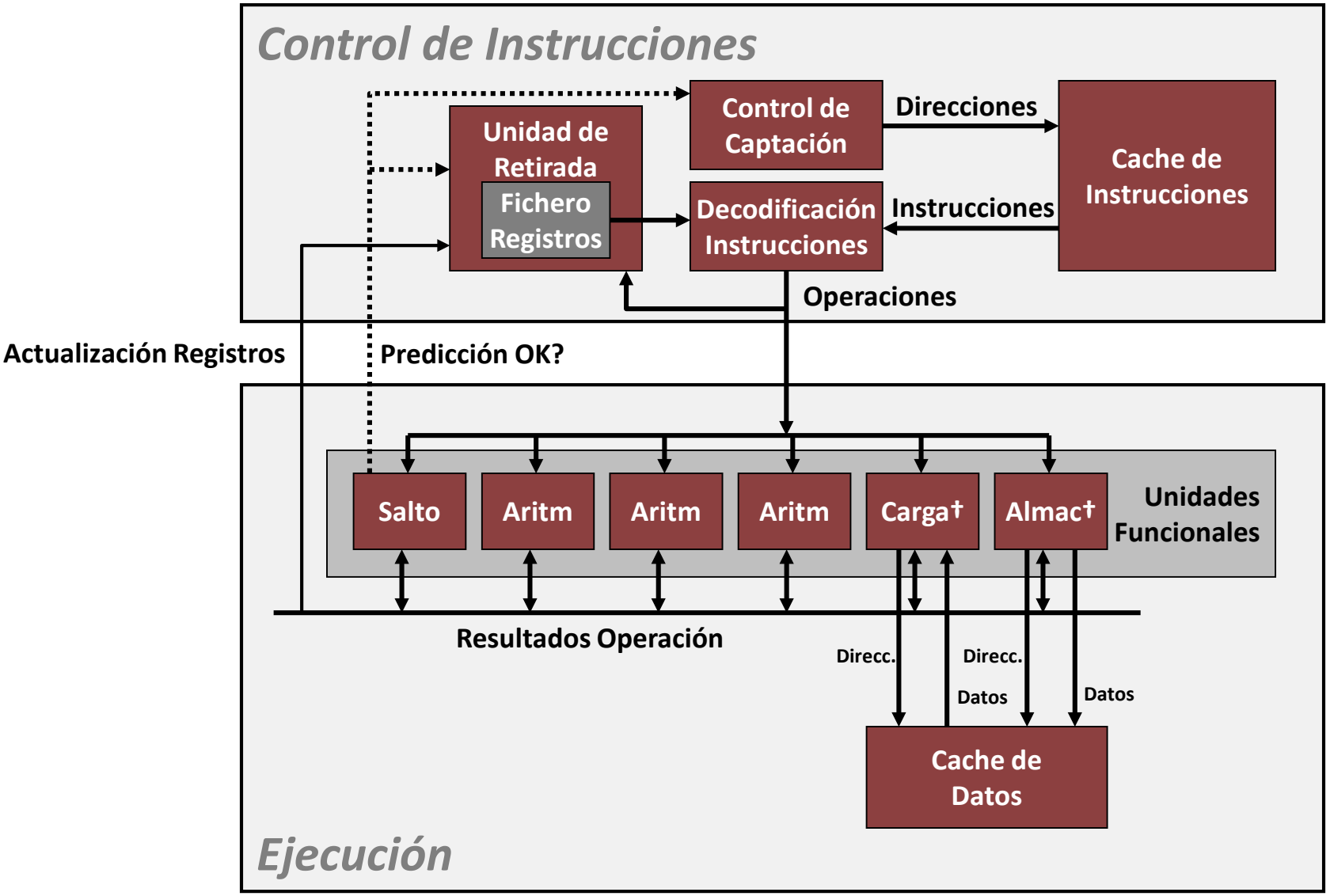
- **Fallos en frío (obligados)**
  - Los fallos en frío ocurren porque la cache empieza vacía y esta es la primera referencia al bloque
- **Fallos por capacidad**
  - Ocurren cuando el conjunto de bloques activos (conjunto de trabajo) es más grande que la cache
- **Fallos por conflicto**
  - Mayoría caches limitan que los bloques a nivel k+1 puedan ir a pequeño subconjunto (a veces unitario) de las posiciones de bloque a nivel k
    - P.ej. Bloque i a nivel k+1 debe ir a bloque (i mod 4) a nivel k (corr. directa)
  - Fallos por conflicto ocurren cuando cache nivel k suficientemente grande pero a varios datos les corresponde ir al mismo bloque a nivel k
    - P.ej. Referenciar bloques 0, 8, 0, 8, 0, 8, ... fallaría continuamente (ejemplo anterior con correspondencia directa)

Memorias Cache

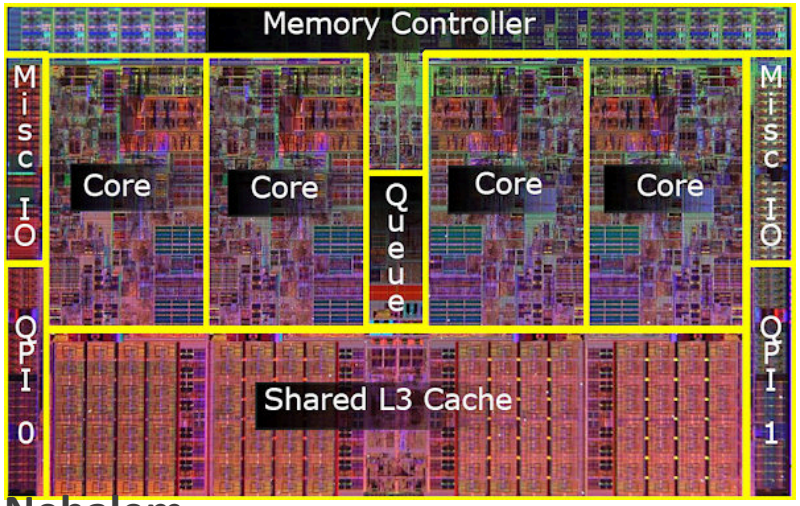
- **Las memorias cache son memorias pequeñas y rápidas basadas en SRAM gestionadas automáticamente por hardware**
  - Retiene bloques de memoria principal accedidos frecuentemente
- **La CPU busca los datos primero en caché**
- **Estructura típica del sistema:**



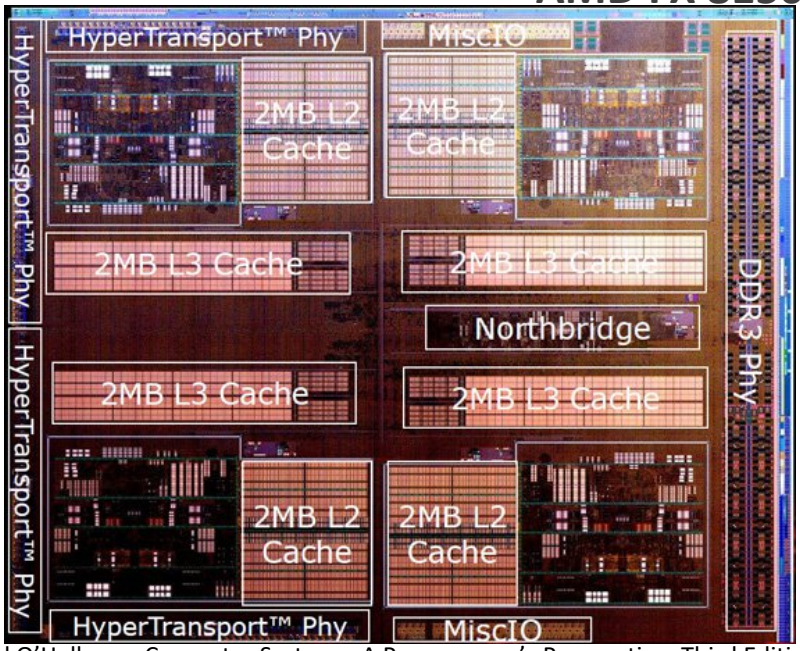
Diseño moderno de CPU



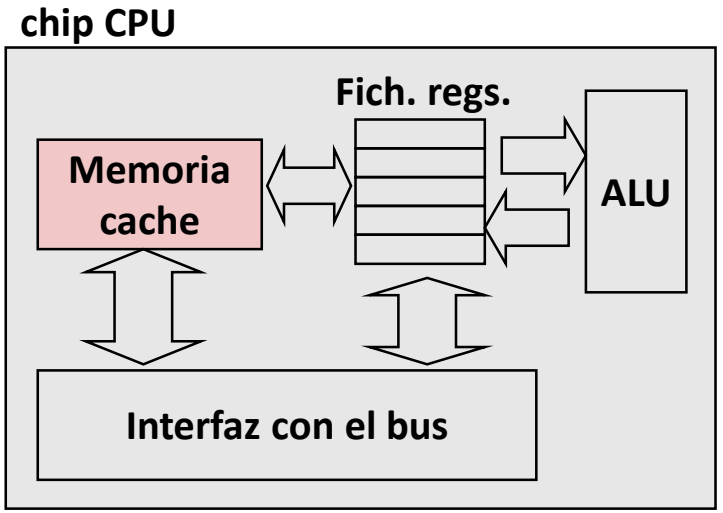
El aspecto que tiene realmente



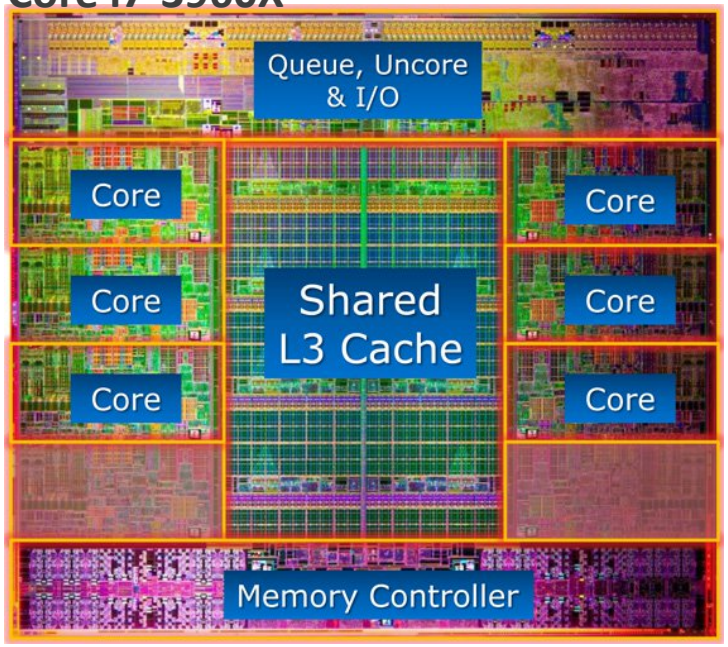
Nehalem



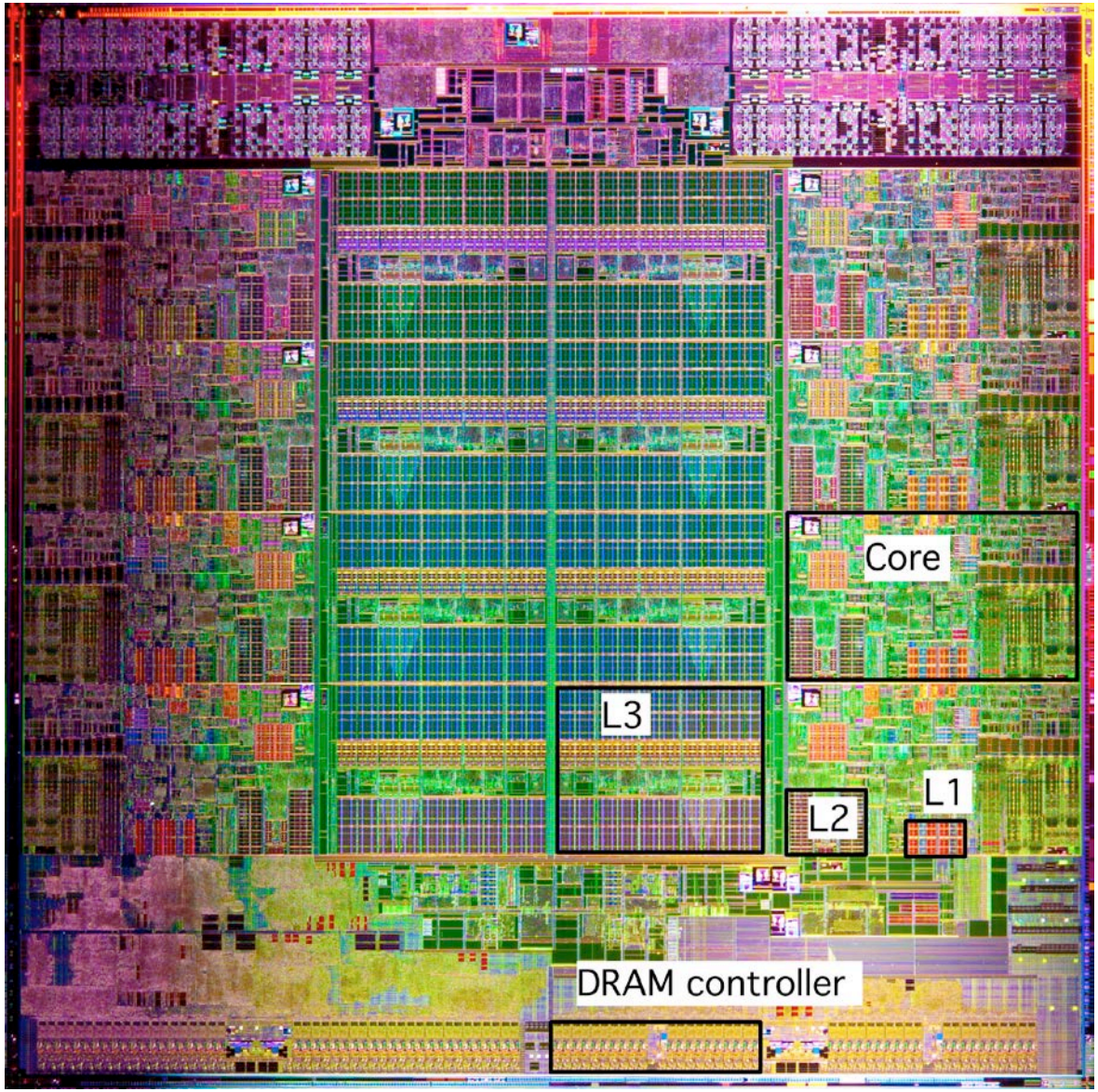
AMD FX 8150



Core i7-3960X



El aspecto que tiene realmente (Cont.)

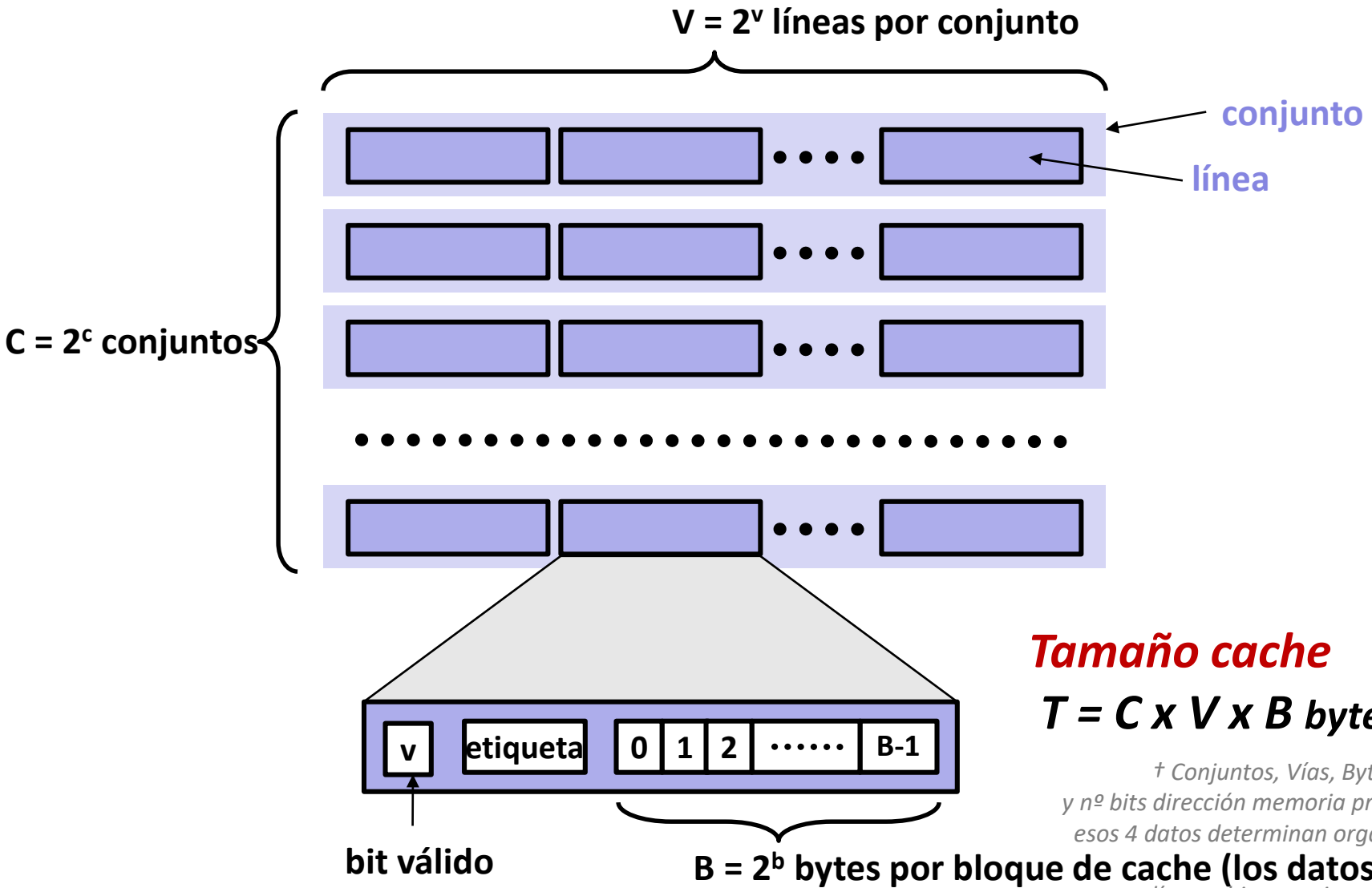


Dado† Procesador Intel Sandy Bridge

L1: 32KB Instrucciones + 32KB Datos
L2: 256KB
L3: 3–20MB

† *Die = dado, Wafer = oblea, que se corta en dados*

Organización General de Cache (C, V, B, m†)



Tamaño cache

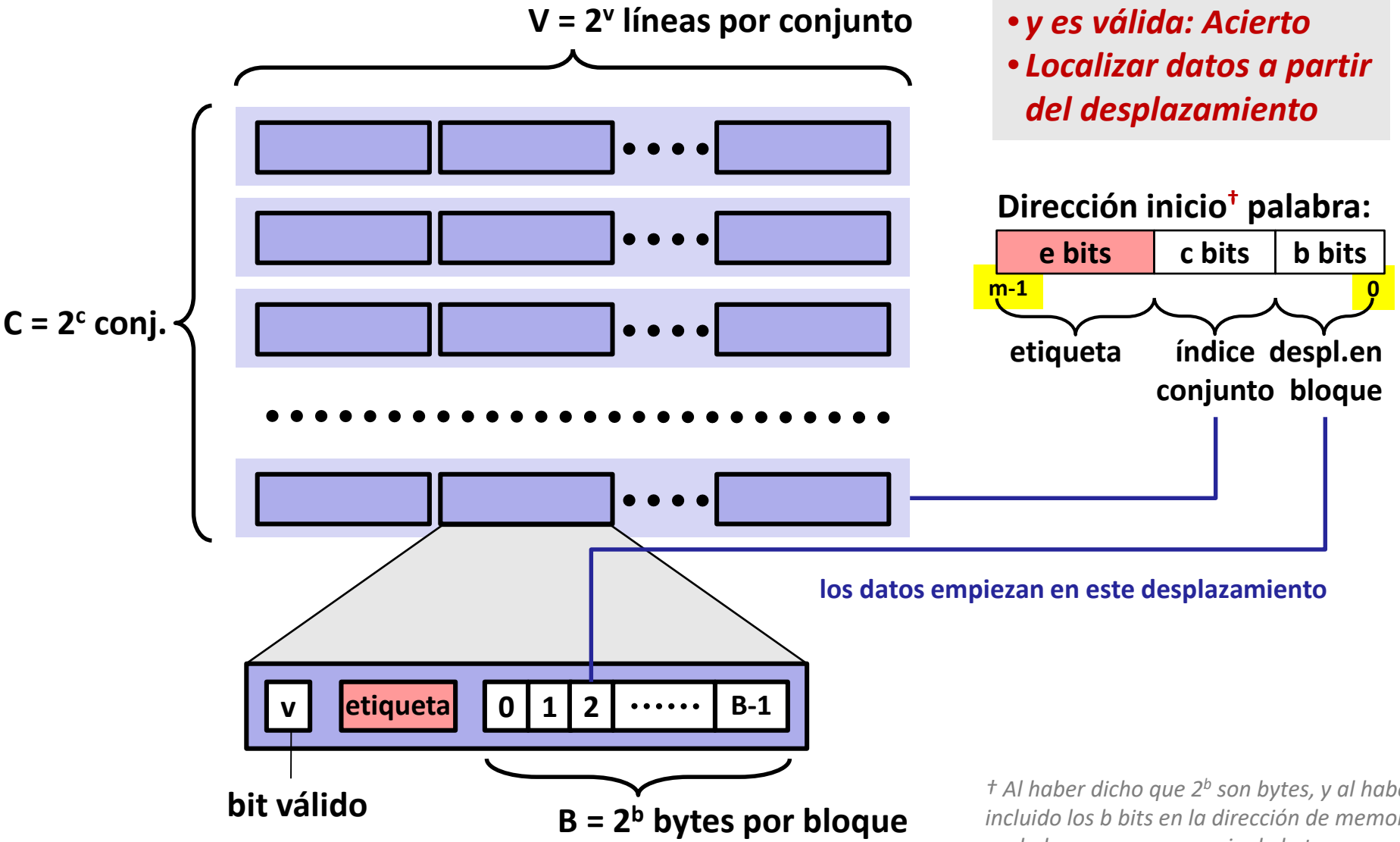
T = C x V x B bytes datos

† Conjuntos, Vías, Bytes (C, V, B) y nº bits dirección memoria principal (m) esos 4 datos determinan organiz. cache

línea = bit v + etiqueta + bloque

Lectura de Cache

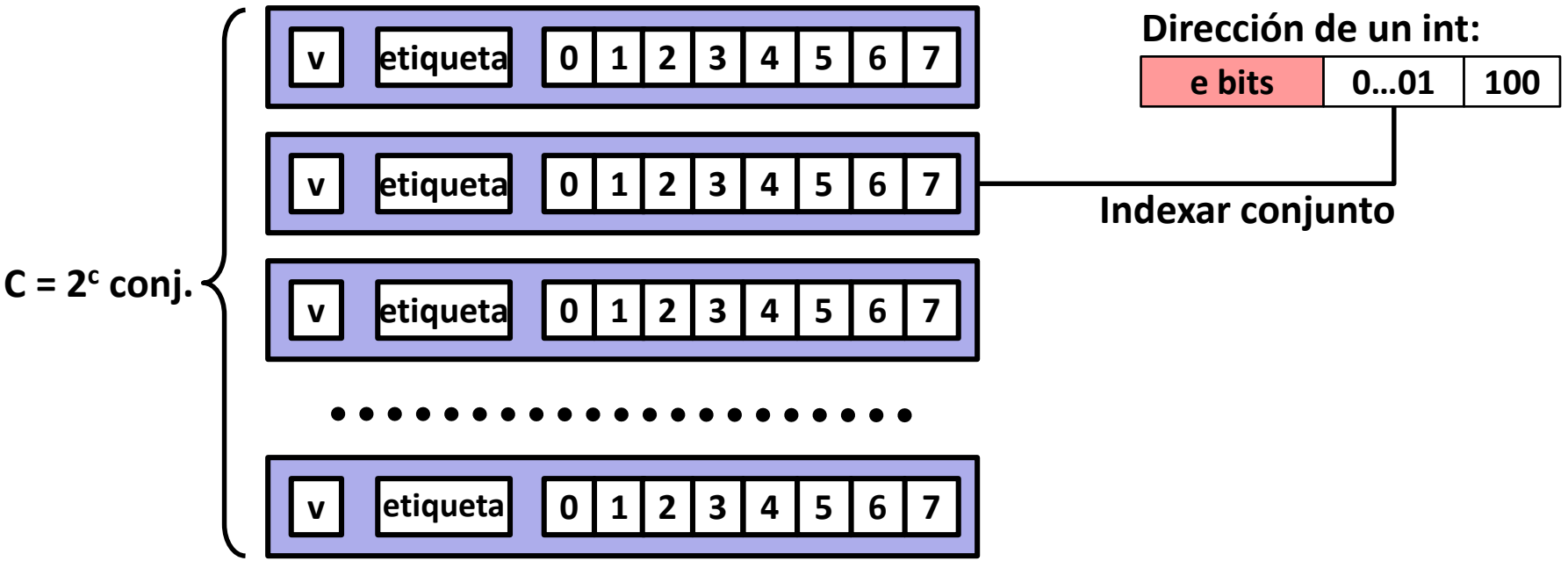
- *Indexar conjunto*
- *Si alguna línea tiene etiqueta coincidente*
- *y es válida: Acierto*
- *Localizar datos a partir del desplazamiento*



† *Al haber dicho que 2^b son bytes, y al haber incluido los b bits en la dirección de memoria, se deduce que es memoria de bytes*

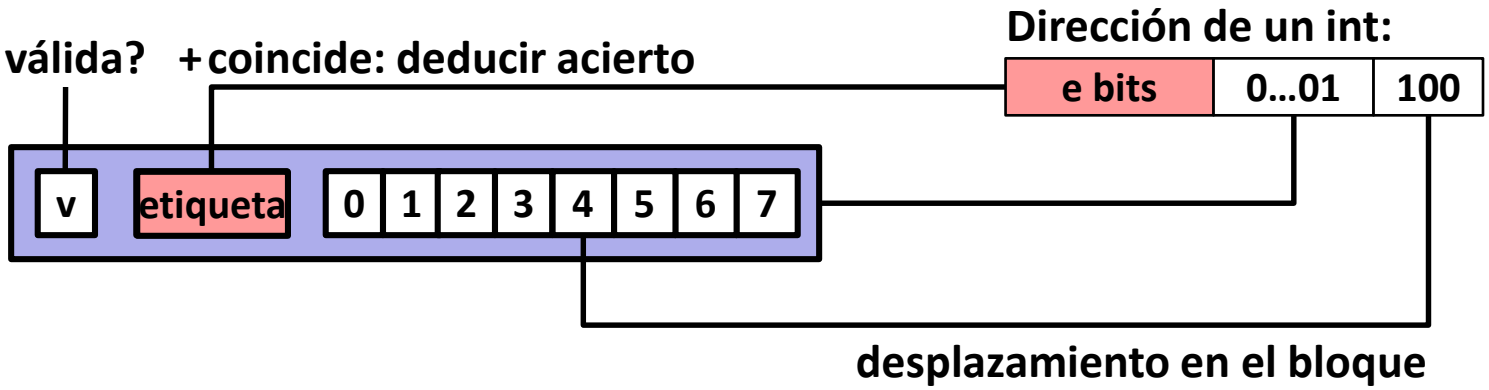
Ej: Cache con Correspondencia Directa (V = 1)

Correspondencia directa: Una línea por conjunto
Suponer: tamaño bloque cache B=8 bytes



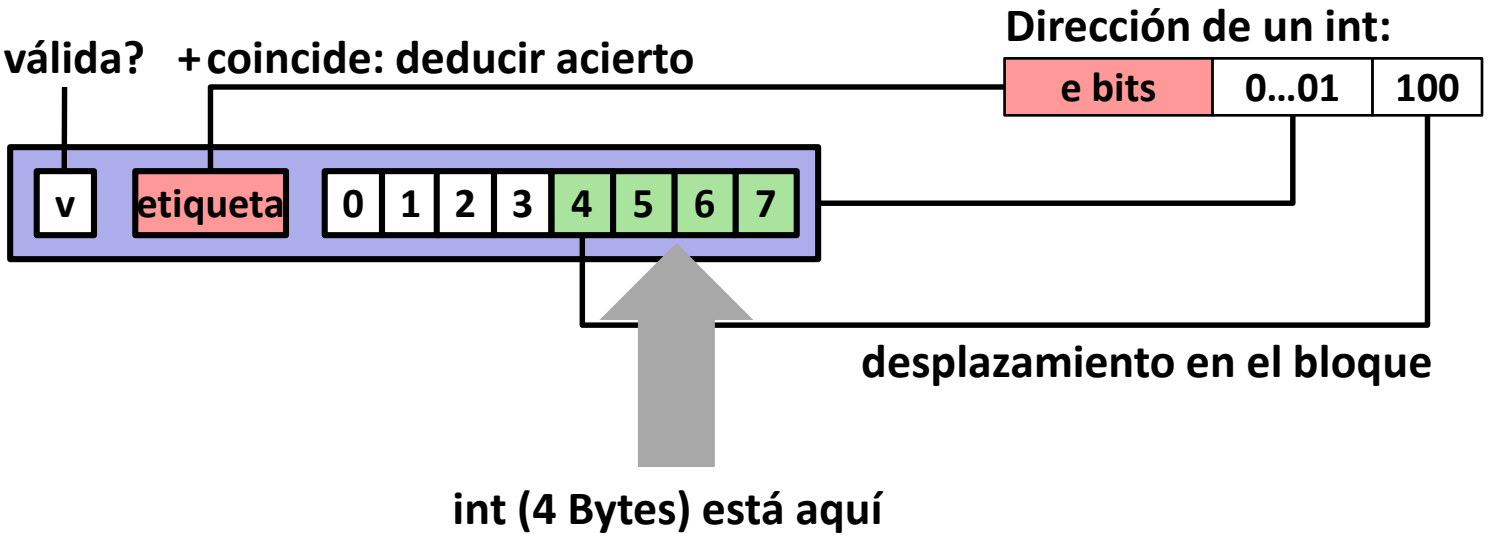
Ej: Cache con Correspondencia Directa (V = 1)

Correspondencia directa: Una línea por conjunto
Suponer: tamaño bloque cache B=8 bytes



Ej: Cache con Correspondencia Directa (V = 1)

Correspondencia directa: Una línea por conjunto
Suponer: tamaño bloque cache B=8 bytes



Si etiqueta no coincide (= fallo): vieja línea desalojada y reemplazada
(nuevos datos y nueva etiqueta)

Simulación cache correspondencia directa

e=1	c=2	b=1
x	xx	x

direcciones 4 bits (espacio dirccnmto tam M=16 bytes)
C=4 conjuntos, V=1 vía (bloq./conj.), B=2 bytes/bloque

Traza de direcciones (lecturas, un byte por lectura):

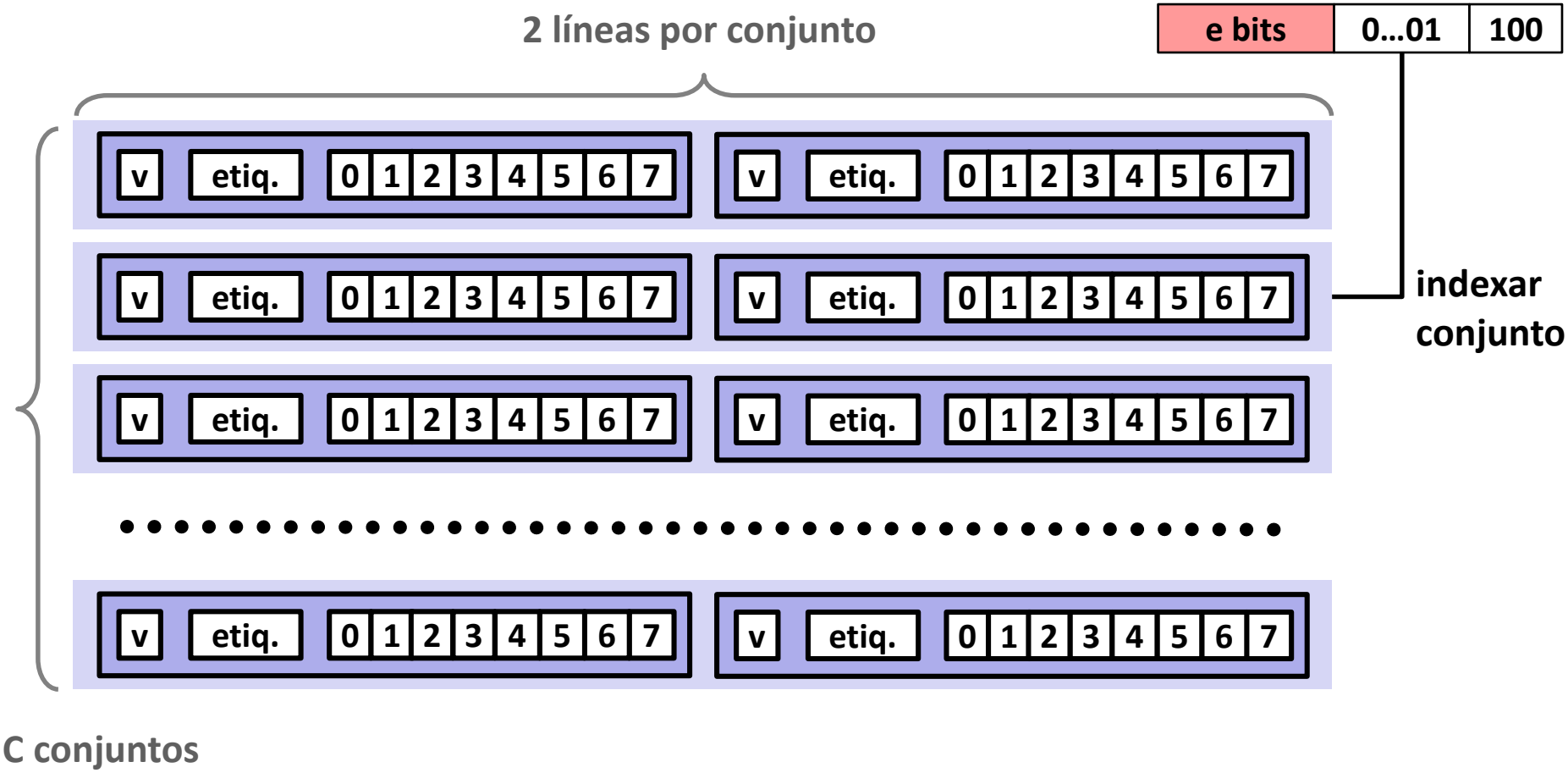
0	**[0000_2],**	fallo
1	**[0001_2],**	acierto
7	**[0111_2],**	fallo
8	**[1000_2],**	fallo
0	**[0000_2]**	fallo

	v	Etiq.	Bloque
Conj. 0	1	0	M[0-1]
Conj. 1	0		
Conj. 2	0		
Conj. 3	1	0	M[6-7]

Cache Asociativa por Conjuntos de V vías (con V=2)

V = 2: Dos líneas por conjunto

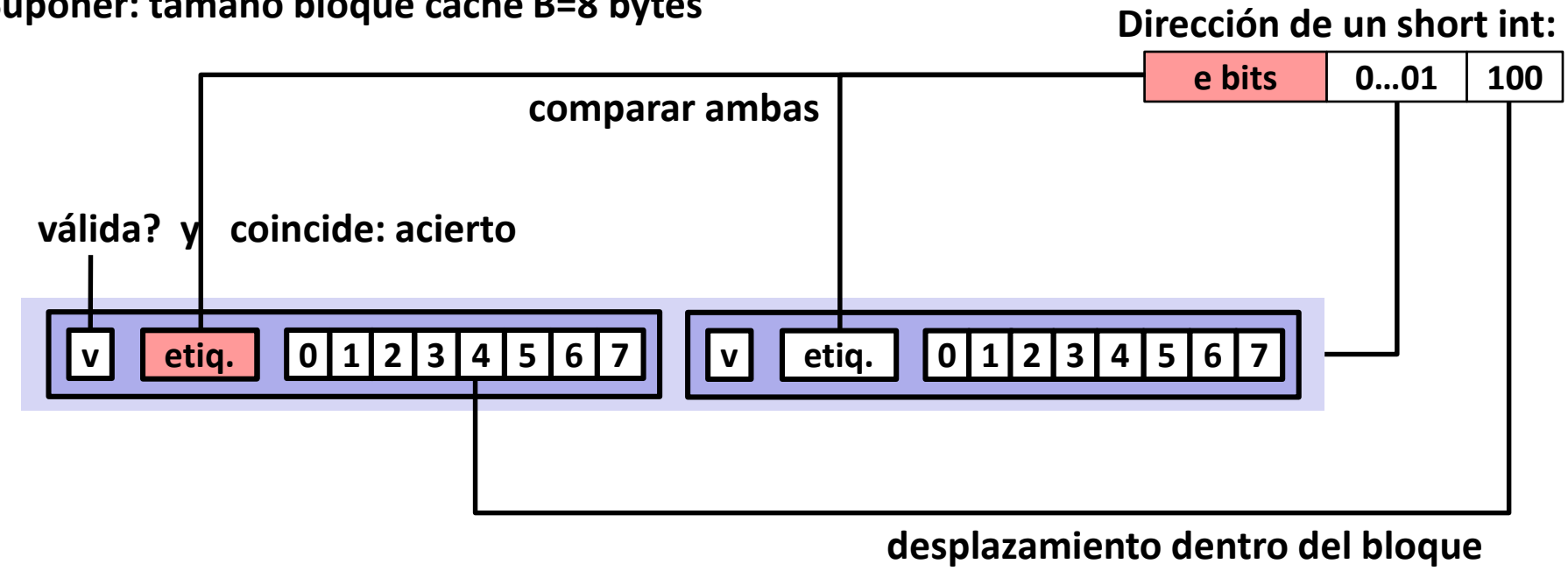
Suponer: tamaño bloque cache B=8 bytes



Cache Asociativa por Conjuntos de V vías (aquí V=2)

V = 2: Dos líneas por conjunto

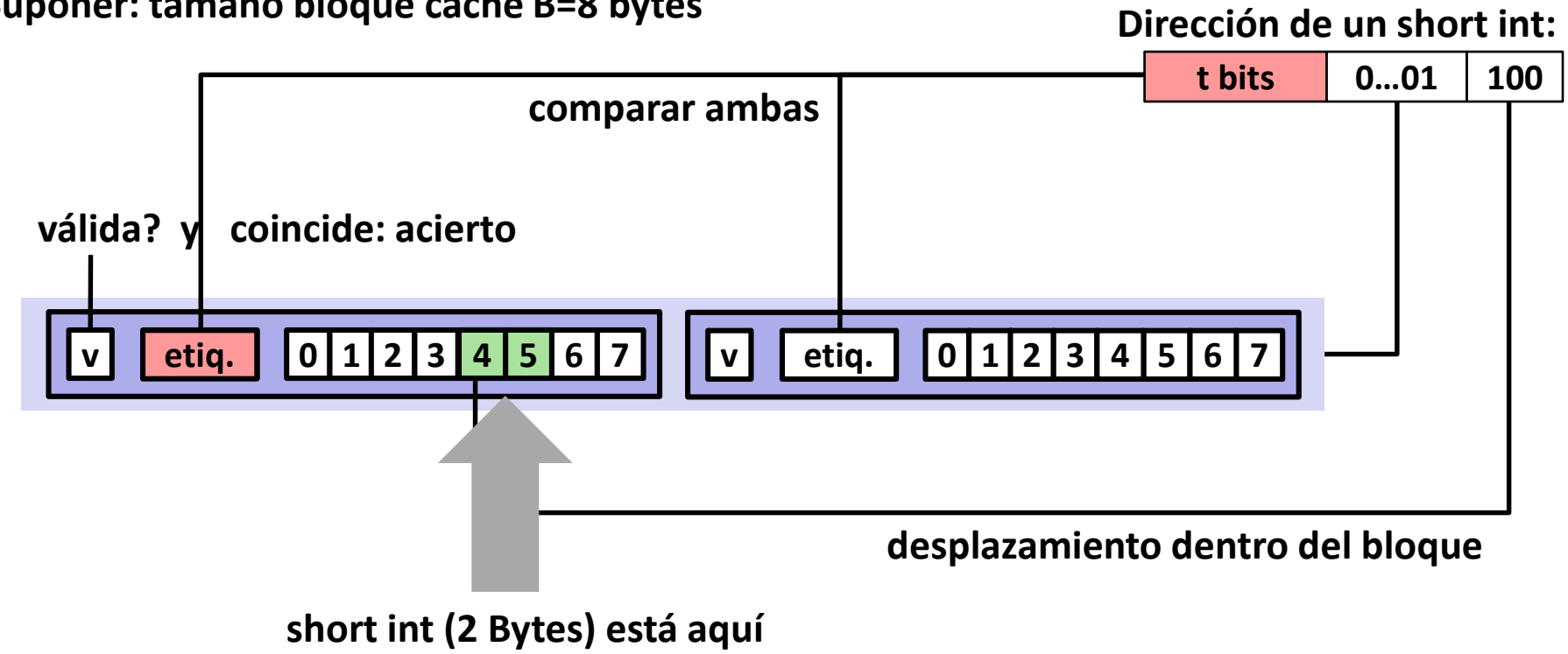
Suponer: tamaño bloque cache B=8 bytes



Cache Asociativa por Conjuntos de V vías (aquí V=2)

V = 2: Dos líneas por conjunto

Suponer: tamaño bloque cache B=8 bytes



Si ninguna coincide o no válida (= fallo):

- Se escoge una línea del conjunto para desalojo y reemplazo
- Políticas de reemplazamiento: aleatoria, menos rec.uso (LRU), …

Simulación cache asociativa conjuntos 2-vías

e=2	c=1	b=1
xx	x	x

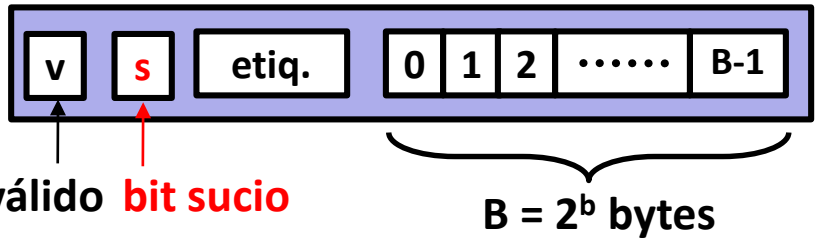
direcciones 4 bits (M=16 bytes)
C=2 conjuntos, V=2 vías (bloq./conj.), B=2 bytes/bloque

Traza de direcciones (lecturas, un byte por lectura):

0	**[0000$_2$],**	fallo
1	**[0001$_2$],**	acierto
7	**[0111$_2$],**	fallo
8	**[1000$_2$],**	fallo
0	**[0000$_2$]**	acierto

	v	Etiq.	Bloque
Conj. 0	1	00	M[0-1]
	1	10	M[8-9]
Conj. 1	1	01	M[6-7]
	0		

¿Qué hay de las escrituras?



- **Múltiples copias de los datos:**
  - L1, L2, L3, Mem. principal, Disco
- **¿Qué hacer en acierto escritura?**
  - Write-through (escribir inmediatamente en la memoria) (Escritura directa)
  - Write-back (diferir escritura hasta reemplazo de la línea) (Escritura diferida)
    - Cada línea caché necesita un bit sucio (=1 si línea difiere de bloque M)
- **¿Qué hacer en fallo escritura?**
  - Write-allocate (cargar y actualizar línea en cache) (Asignación en Escritura)
    - Conveniente si van a haber más escrituras a ese bloque
  - No-write-allocate (escribir directo a memoria, sin cargar en cache)
- **Usual**
  - Write-through + No-write-allocate (Escritura directa sin Asignación en Escritura)
  - **Write-back + Write-allocate** **(Post-Escritura con Asignación en Escritura)**

¿Por qué indexar con los bits intermedios?

Correspondencia directa: Una línea por conjunto
Suponer: tamaño bloque cache B=8 bytes

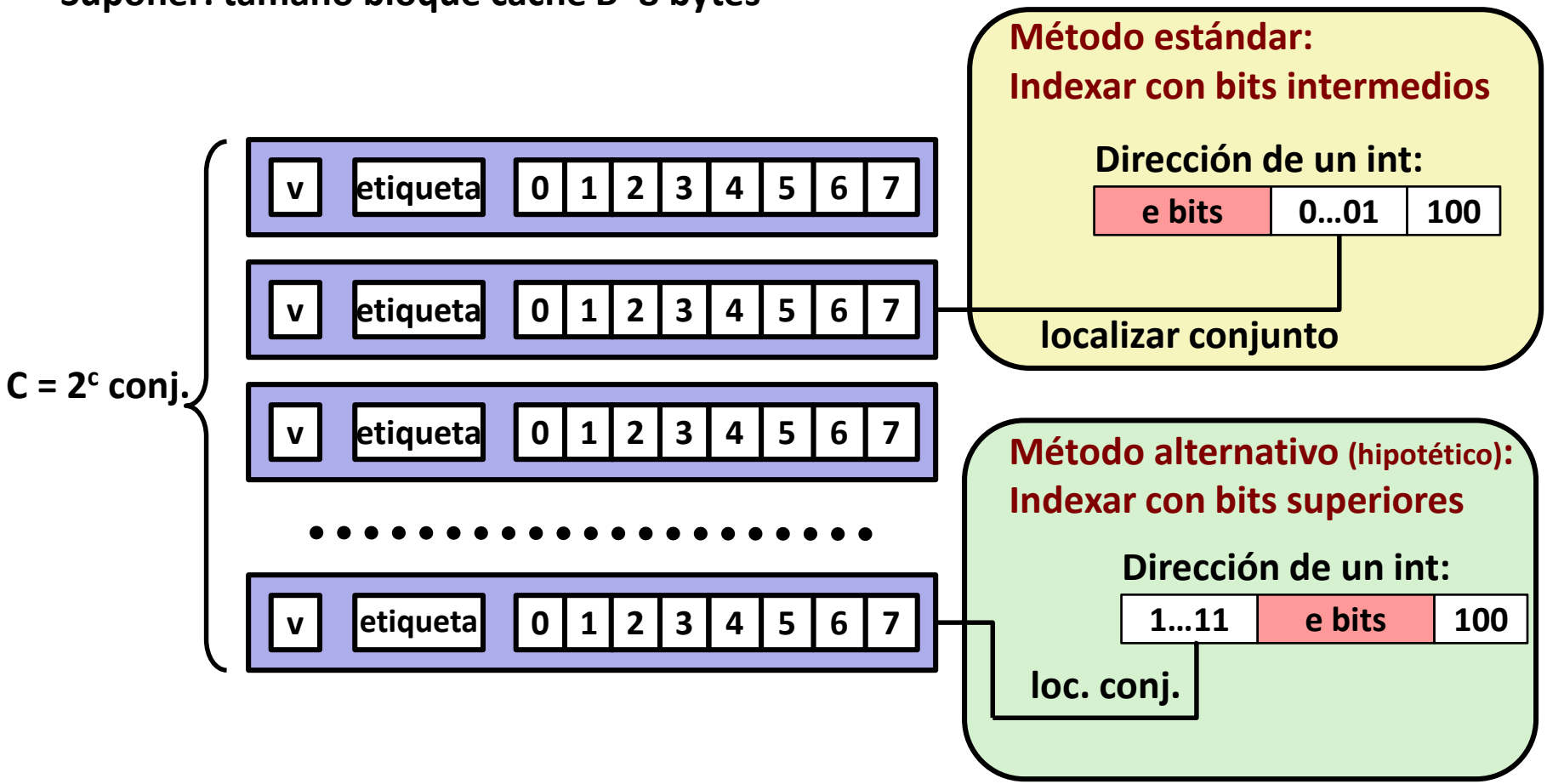
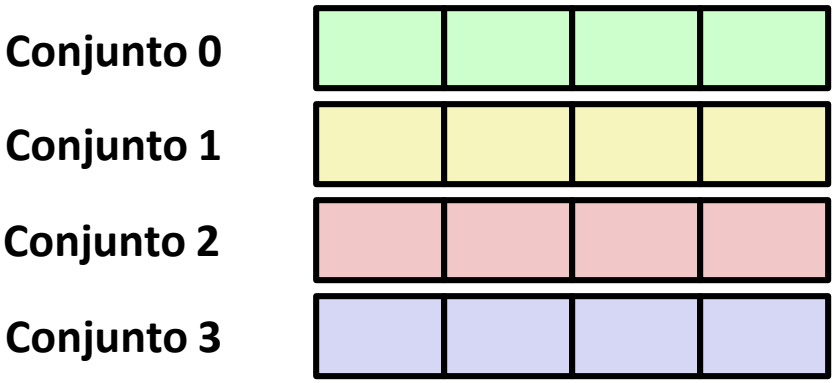
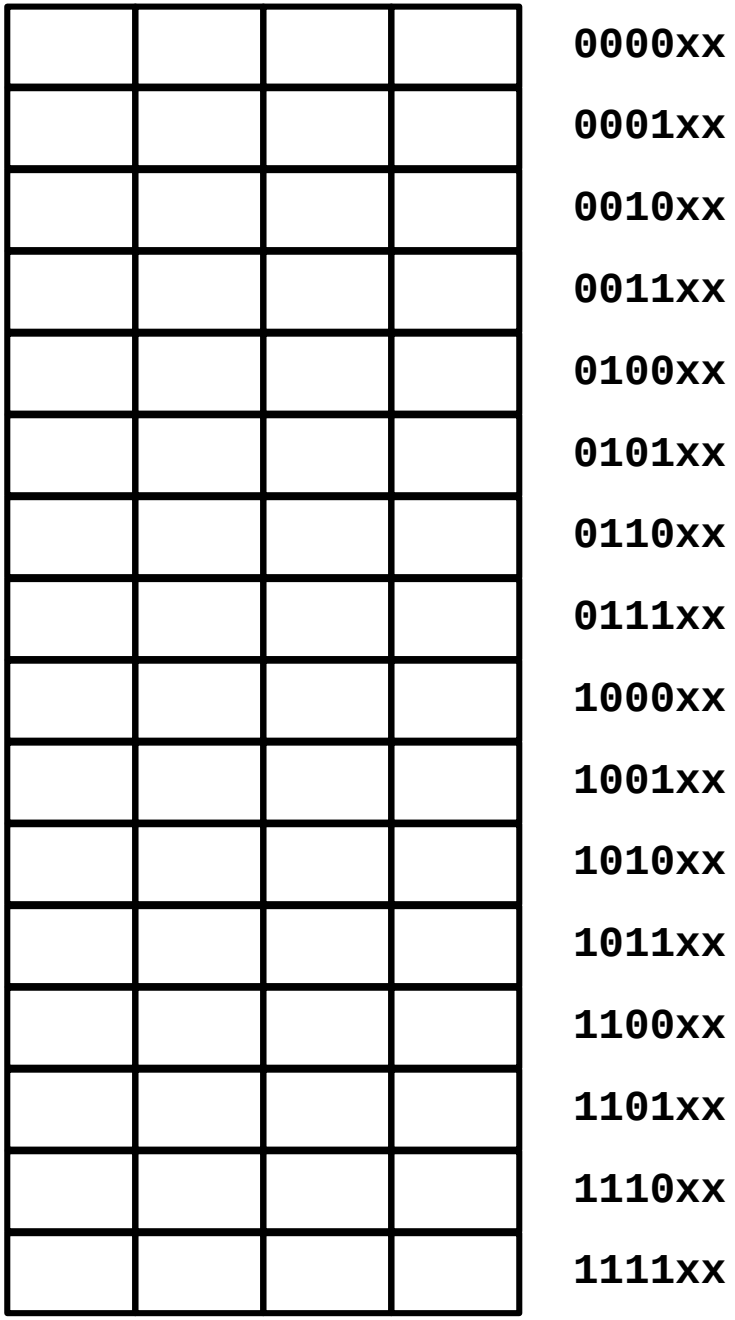


Ilustración de los métodos de indexación

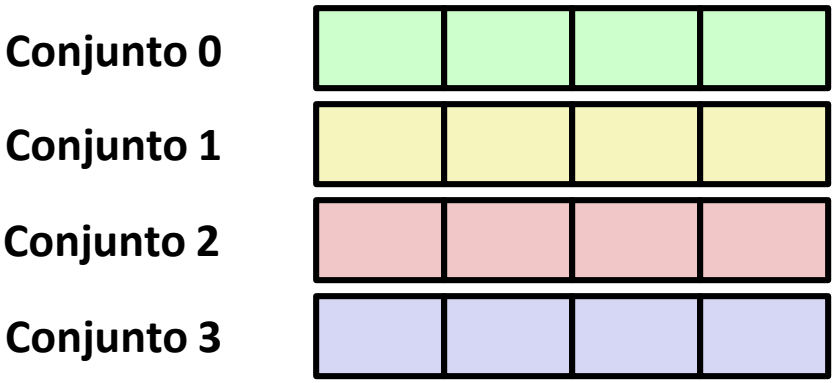
- **Memoria de 64 bytes**
  - Direcciones de 6 bits
- **Cache de 16B, corresp. directa**
- **T. bloque B=4** (⇒C=4 conj. ¿Por qué?)
- **2 bits etiq., 2 bits índ., 2 bits despl.**

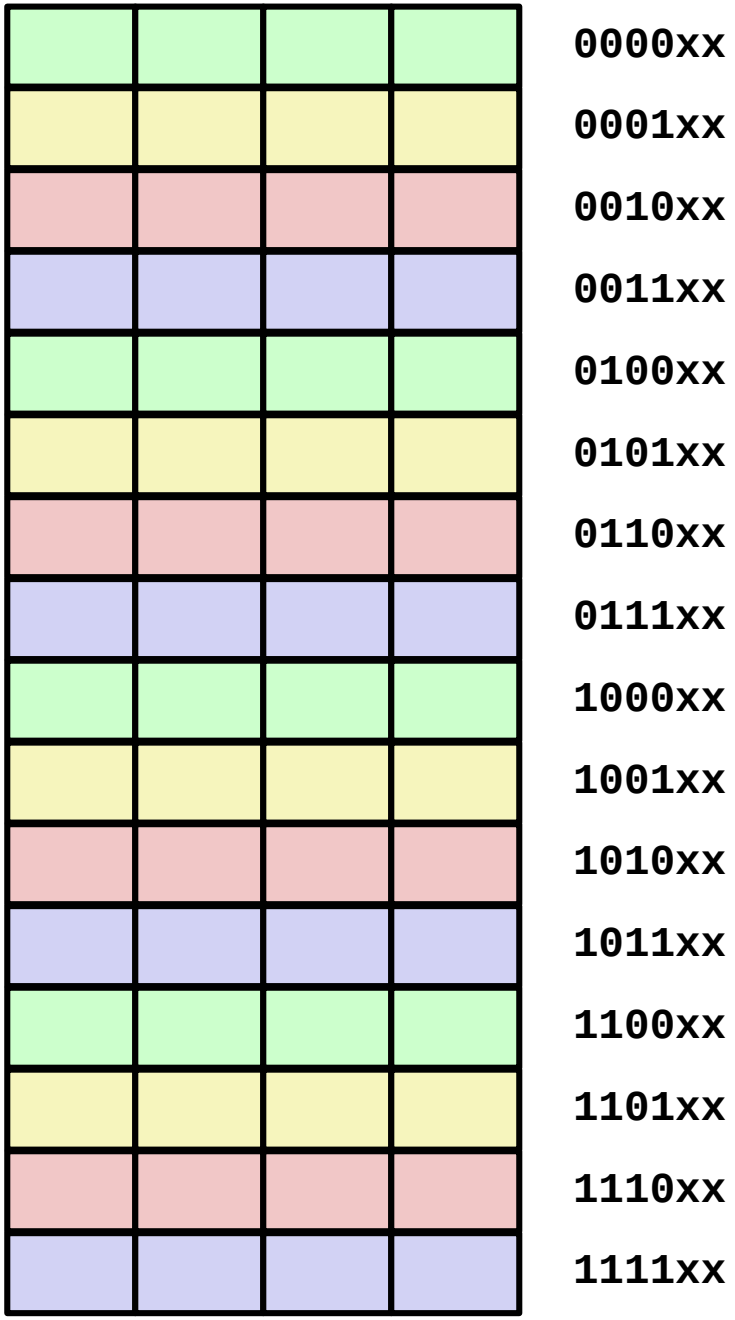




Indexado c/bits intermedios

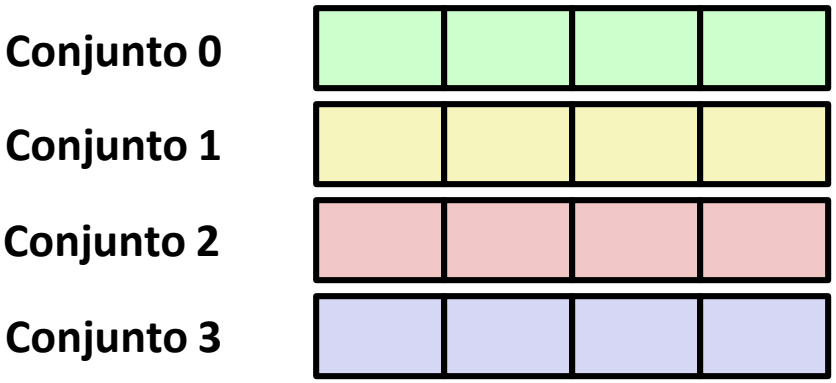
- Direcciones de la forma TTSSBB
  - TT bits etiqueta
  - SS bits índice conjunto
  - BB bits desplazamiento bloque
- Hace buen uso de localidad espacial

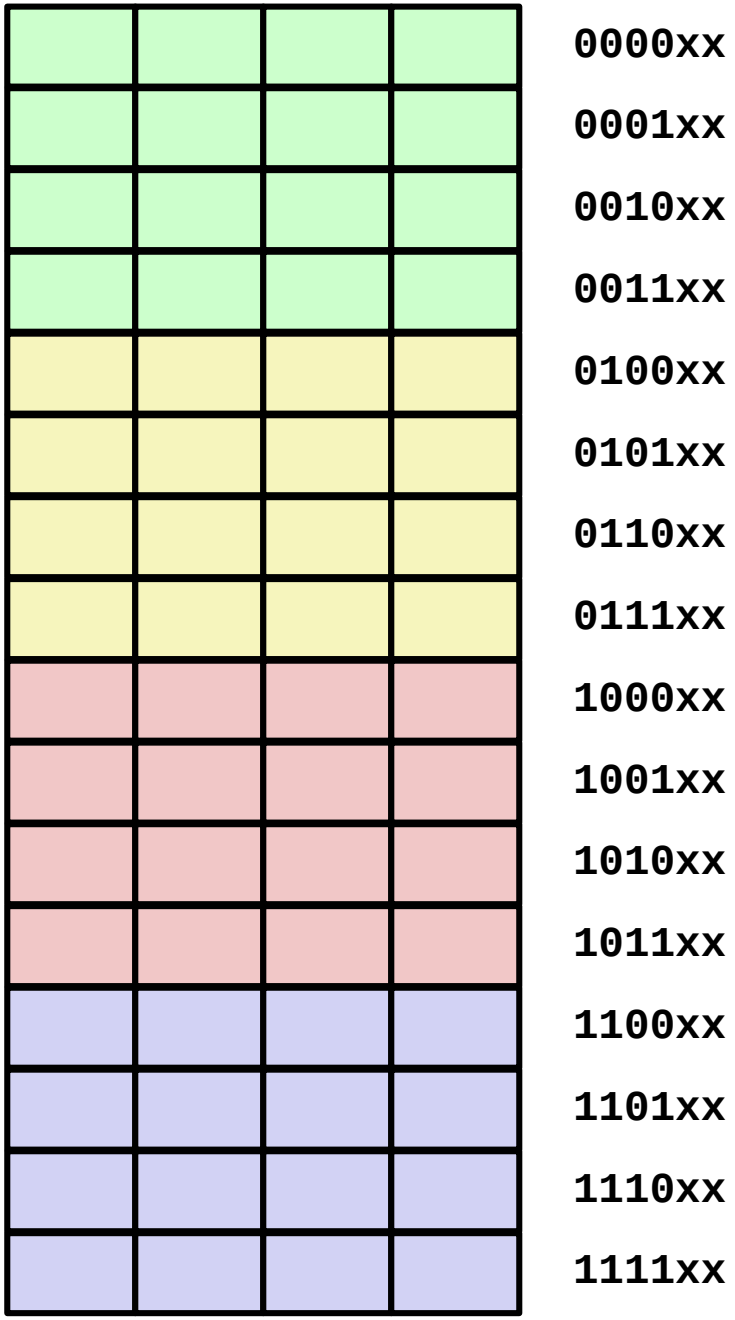




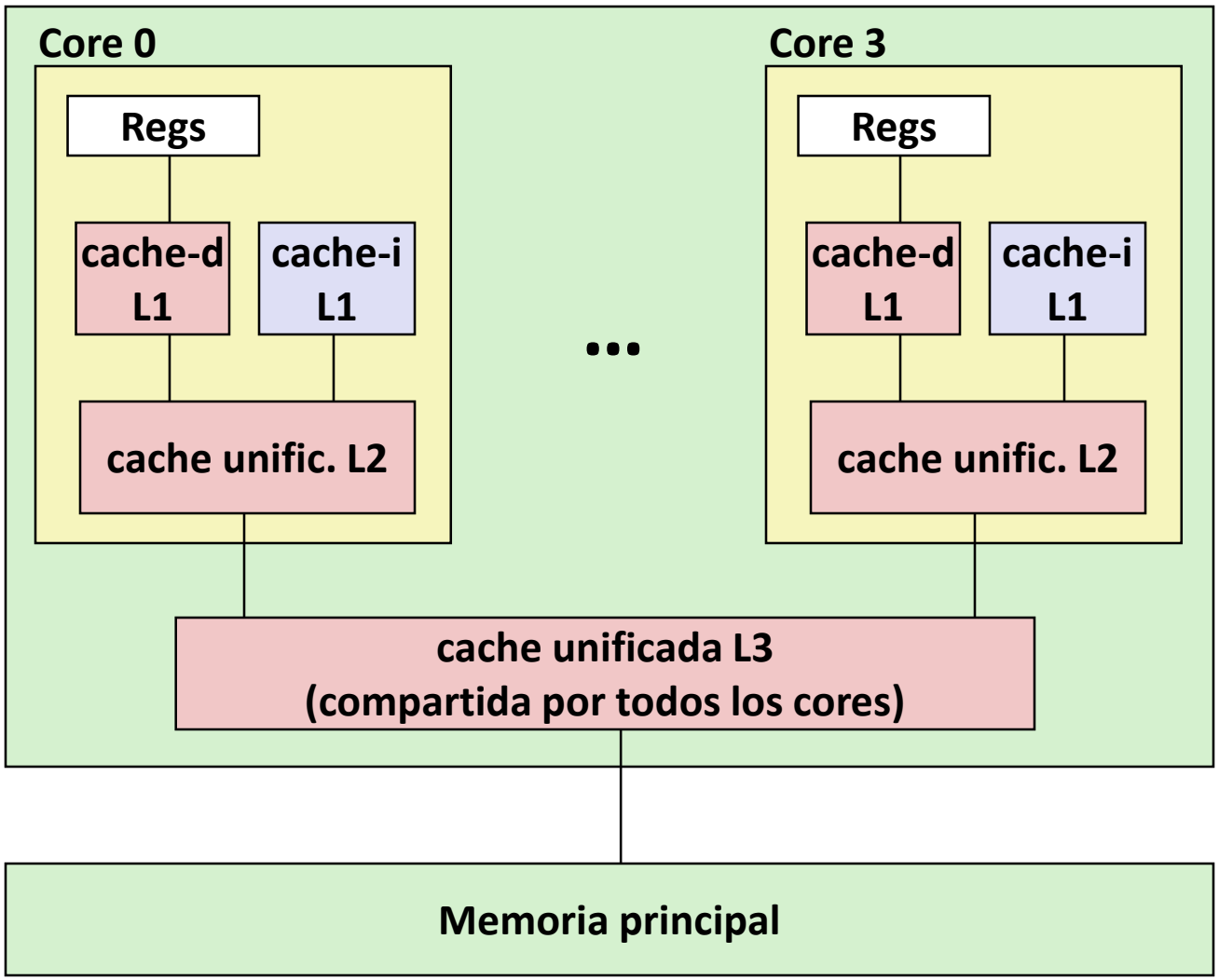
Indexado c/bits superiores

- **Direcciones de la forma SSTTBB**
  - **SS** bits índice conjunto
  - **TT** bits etiqueta
  - **BB** bits desplazamiento bloque
- **Programa con alta localidad espacial generaría muchos conflictos** (fallos por conflicto)





Jerarquía de caches del Intel Core i7



cache-i y cache-d L1:
32 KB, 8-vías,
Acceso: 4 ciclos

cache unificada L2:
256 KB, 8-vías,
Acceso: 10 ciclos

cache unificada L3:
8 MB, 16-vías,
Acceso: 40-75 ciclos

Tamaño de bloque: 64 B en todas las caches

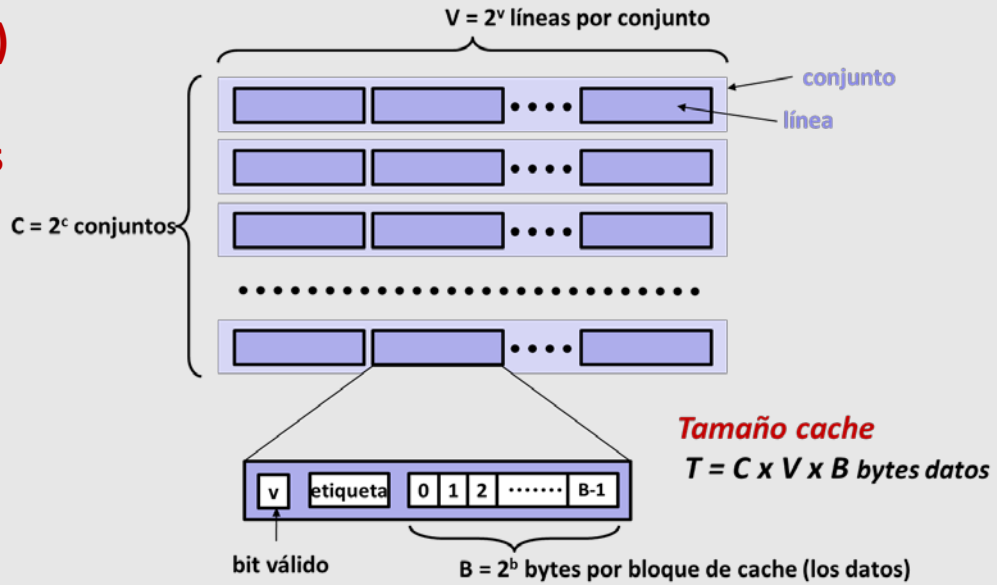
…pero el propio Intel prefiere Paquete

† Processor package debería traducirse por Empaquetamiento

Ejemplo: cache de datos L1 del Core i7

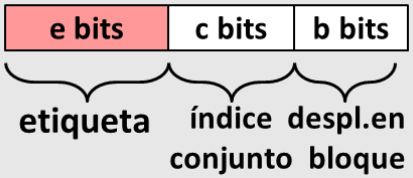
32 KB 8-vías (asoc.conj.)
64 bytes/bloque
47 bit rango direcciones

B = , b =
V = , v =
C = , c =
T =



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Dirección inicio palabra:



despl. bloque: _ bits
índice conj.: _ bits
etiqueta: _ bits

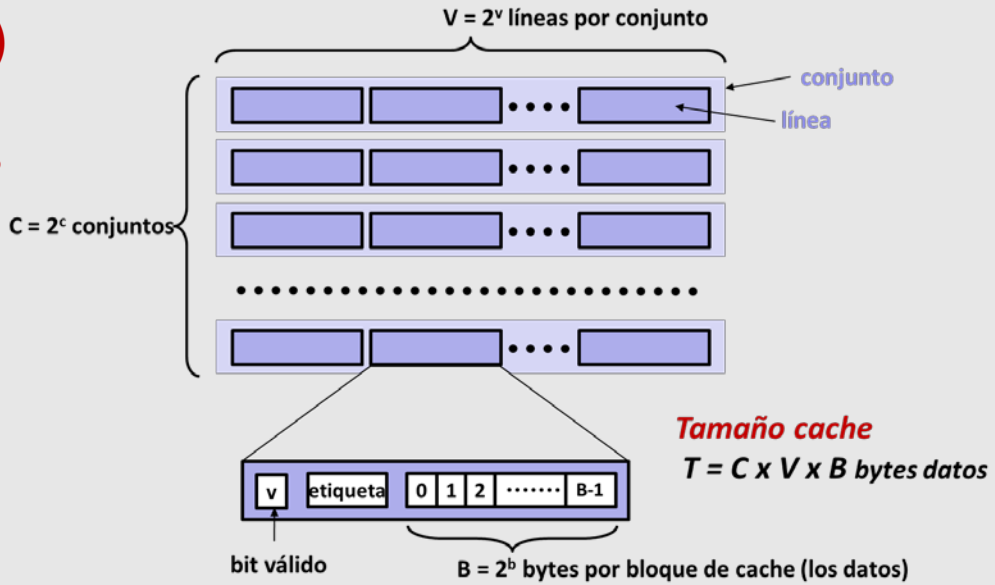
Dirección de Pila:
0x00007f7262a1e010

despl. bloque: 0x??
índice conj.: 0x??
etiqueta: 0x??

Ejemplo: cache de datos L1 del Core i7

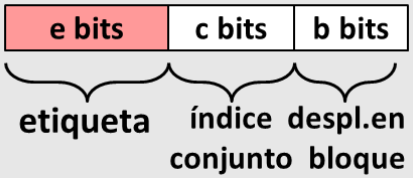
32 KB 8-vías (asoc.conj.)
64 bytes/bloque
47 bit rango direcciones

B = 64, b = 6
V = 8, v = 3
C = 64, c = 6
T = 64 x 64 x 8 = 32,768



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Dirección inicio palabra:



despl. bloque: 6 bits
índice conj.: 6 bits
etiqueta: 35 bits

Dirección de Pila:
0x00007f7262a1e010

0000 0001 0000

despl. bloque: 0x10
índice conj.: 0x0
etiqueta: 0x7f7262a1e

Cache: resumen de políticas de colocación

- **Organización (C,V,B,m)**
  - La caché tiene 2^c x 2^v = 2^{c+v} líneas. Un bloque tiene 2^b bytes. Una dirección física tiene m bits. La MP tiene 2^m bytes (2^{m-b} bloques).
- **Correspondencia directa**
  - Bloque ***i*** de MP ⇒ línea ***i* mod 2^l** de cache (L líneas=2^l=2^{c+v}=2^c con v=0)
- **Correspondencia totalmente asociativa**
  - Bloque ***i*** de MP ⇒ cualquier línea de cache
- **Correspondencia asociativa por conjuntos**
  - Bloque ***i*** de MP ⇒ conjunto ***i* mod 2^c** de cache (cualquier línea del conjunto)

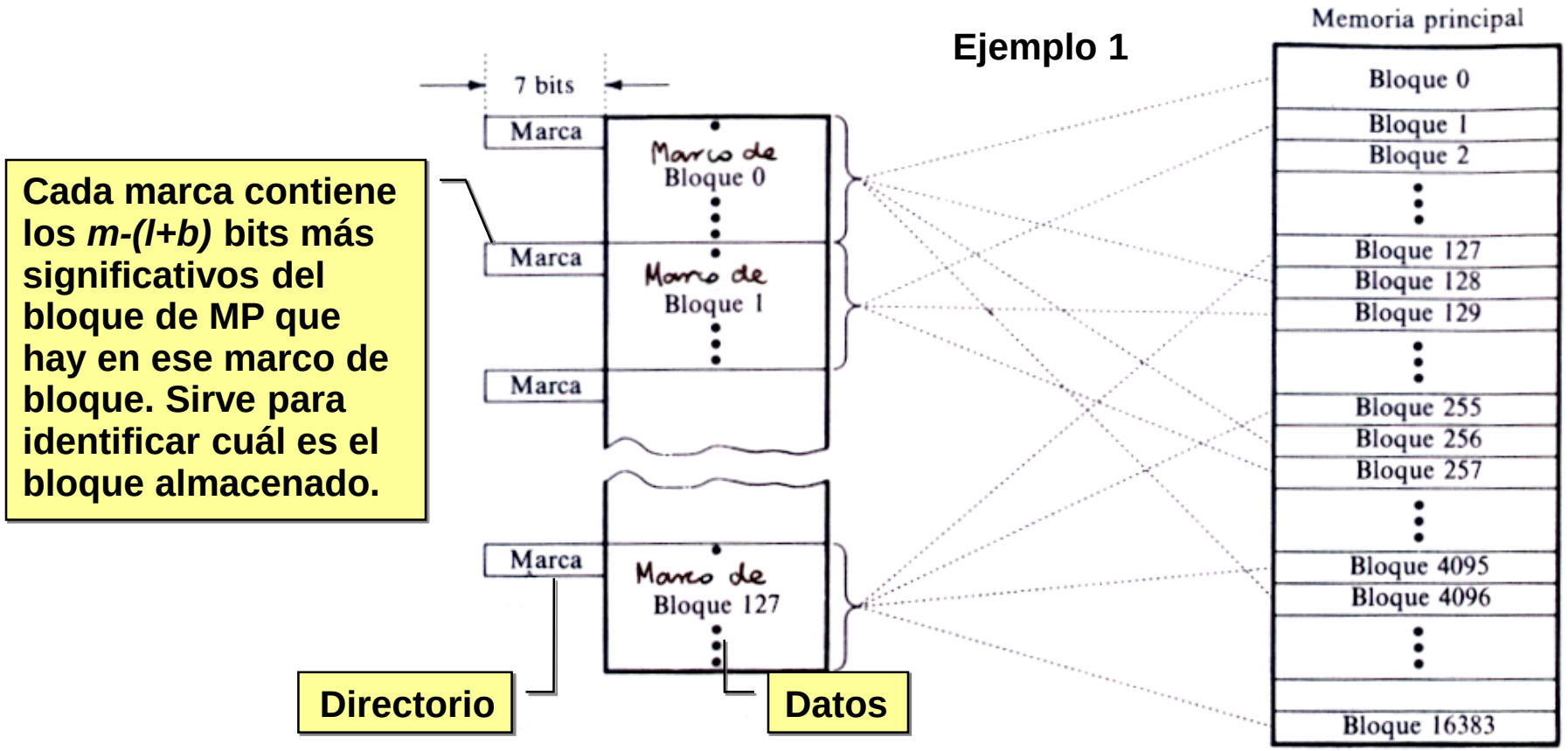
- **Consideraremos el "Ejemplo 1":**
  - Tamaño de caché: 2K bytes. ⇒ c+v+b=11
  - 16 bytes por bloque. b=4 ⇒ c+v=7, **128 líneas en cache**
  - Memoria principal máx: 256K bytes. ⇒ m=18, **16K bloques en MP**

  - (CxV=128, B=16, m=18), c+v=7, b=4, c+v+b=11, CxVxB=2K

Cache: política de colocación

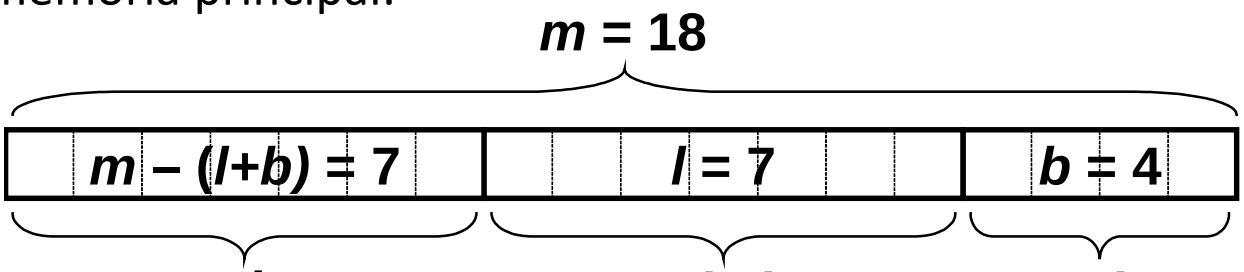
- **Correspondencia directa**
  - Bloque ***i*** de MP ⇒ línea ***i* mod 2^l** de caché.
  - A cada línea le corresponde sólo un subconjunto de bloques de MP.

Ejemplo 1



Cache: política de colocación

- Dirección de memoria principal:



Identifica cada uno de los bloques de MP asociados a un mismo marco de bloque de caché

Marco de bloque donde debe residir en caché este bloque de MP

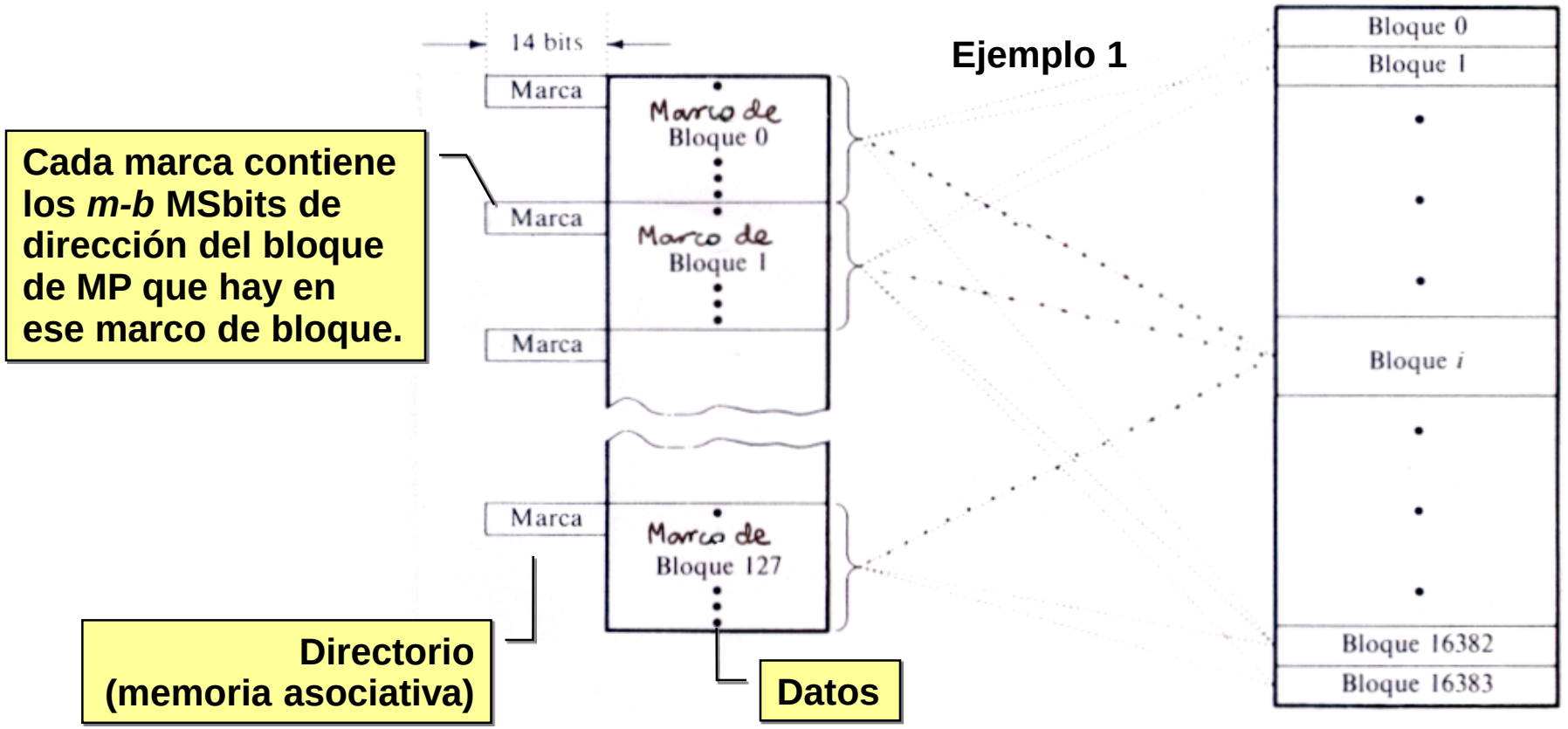
Palabra dentro del bloque

✓ **Simplicidad y bajo coste.**

✘ **Si dos o más bloques, utilizados alternativamente, corresponden al mismo marco de bloque ⇒ el índice de aciertos se reduce drásticamente.**

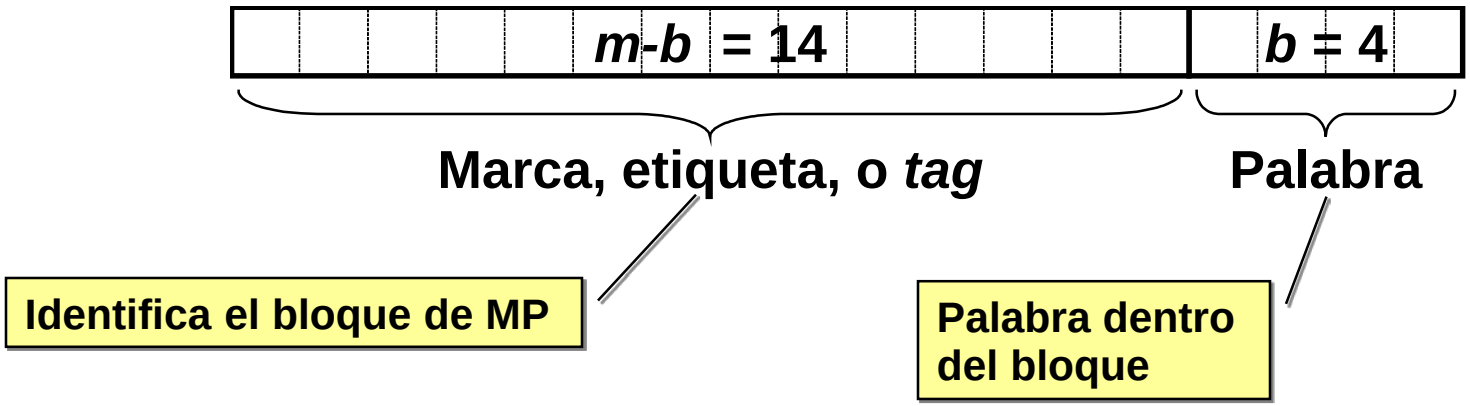
Cache: política de colocación

- **Correspondencia totalmente asociativa**
  - Un bloque de MP puede residir en cualquier marco de bloque de caché.
  - Cuando se presenta una solicitud a la caché, todas las marcas se comparan **simultáneamente** para ver si el bloque está en la caché.



Cache: política de colocación

- Dirección de memoria principal:



✓ **Flexible. Permite cualquier combinación de bloques de MP en la caché. Elimina en gran medida conflictos entre bloques.**

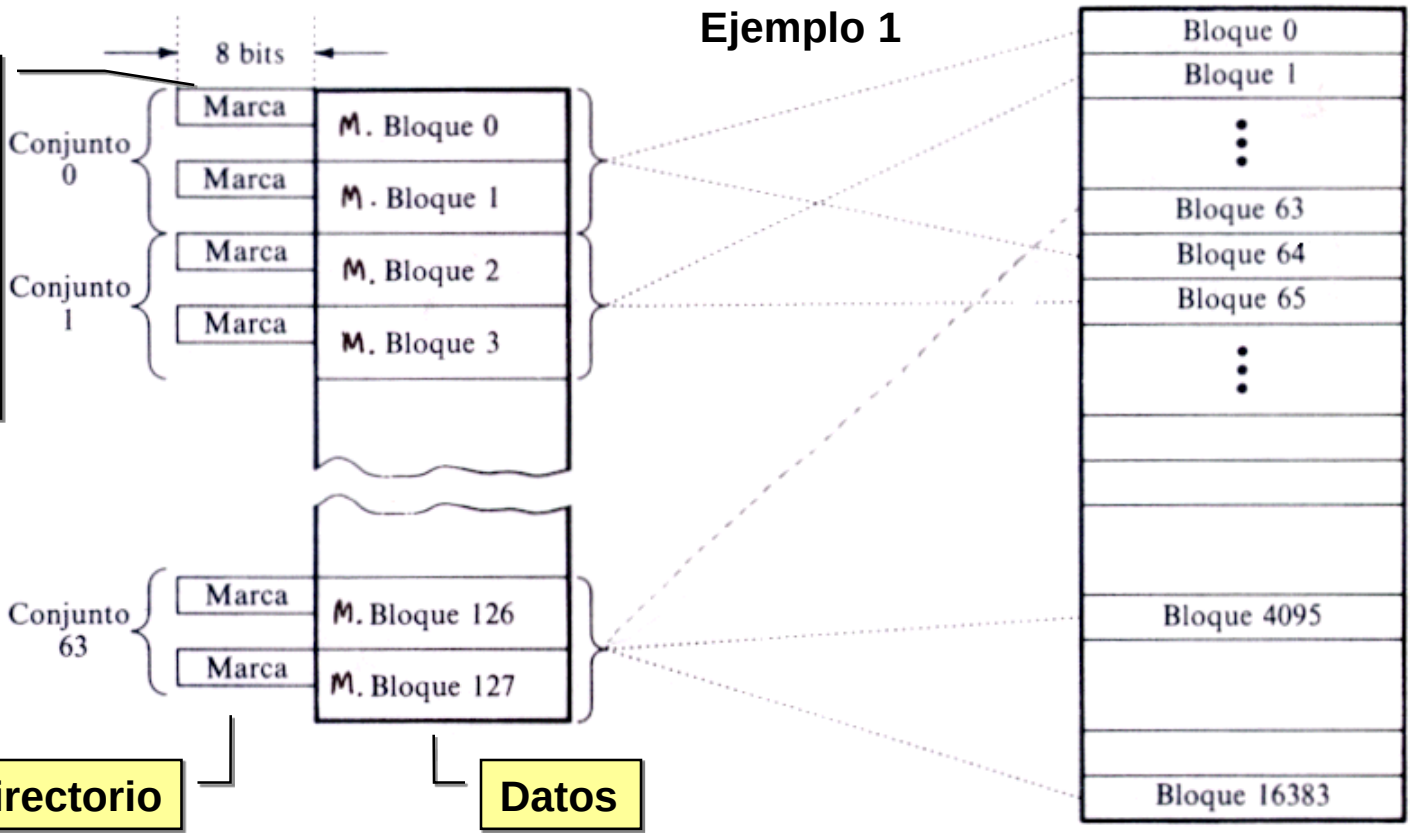
✗ **Compleja y costosa de implementar (por la memoria asociativa).**

Cache: política de colocación

- **Correspondencia asociativa por conjuntos**
  - La caché se subdivide en 2^c conjuntos disjuntos
    - 2^v marcos de bloque / conjunto
  - Bloque ***i*** de MP $\Rightarrow$ conjunto ***i* mod 2^c** de caché. Dentro de ese conjunto puede estar en cualquier marco de bloque.
  - Hay dos fases en el acceso a caché:
    - Selección directa del conjunto donde puede estar ese bloque.
    - Búsqueda asociativa (dentro del conjunto) de la marca.
  - A la correspondencia asociativa por conjuntos con 2^v marcos de bloque / conjunto también se le llama **correspondencia asociativa de 2^v vías**.
    - La vía *i* está formada por todos los marcos de bloque de la caché que ocupan el lugar *i*-ésimo dentro de su conjunto.
    - Completar enunciado del Ejemplo 1 fijando v = 1, 2^v = 2 vías.

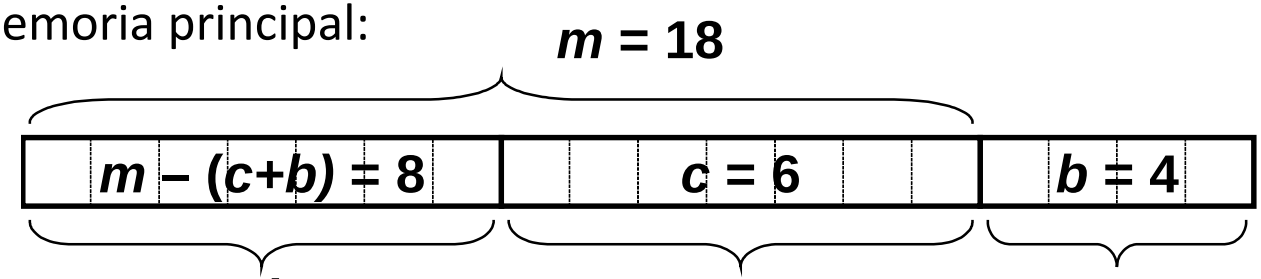
Cache: política de colocación

Cada marca contiene los *m-(c+b)* MSbits de dirección del bloque de MP que hay en ese marco de bloque.



Cache: política de colocación

- Dirección de memoria principal:



c = 0 (1 conjunto) ⇒ corresp. totalmente asociativa.

c = n (1 marco de bloque / conjunto) ⇒ corresp. directa.

0 < c < n ⇒ se pretende reducir el coste de la totalmente asociativa manteniendo un rendimiento similar ⇒ es la técnica más utilizada.

✓ Resultados experimentales demuestran que un tamaño de conjunto de 2 a 16 marcos de bloque funciona casi tan bien como una corresp. totalmente asociativa con un incremento de coste pequeño respecto de la corresp. directa.

Métricas para prestaciones de cache

- **Tasa de Fallo**
  - Fracción de referencias a memoria no encontradas en caché (fallos / accesos) = 1 – tasa de acierto
  - Valores típicos (en porcentaje):
    - 3-10% para L1
    - puede ser bastante pequeño (por ejemplo, <1%) para L2, dependiendo del tamaño, etc.
- **Tiempo en Acierto**† (tiempo de acceso en caso de acierto)
  - Tiempo para entregar una línea de cache al procesador
    - incluye el tiempo para determinar si la línea está en cache
  - Valores típicos :
    - 4 ciclos reloj para L1
    - 10 ciclos reloj para L2
- **Penalización por Fallo**
  - Tiempo adicional requerido debido a un fallo
    - típicamente 50-200 ciclos para M.principal (Tendencia: ¡aumentando!)

† *Hit time*

Reflexionemos sobre esos valores

- **Enorme diferencia entre acierto y fallo**
  - Podría llegar a 100x, si solo L1 y Memoria principal
  - Could be 100x, if just L1 and main memory

- **¿Verosímil que 99% acierto es doble de bueno que 97%?**
  - Considerar este ejemplo simplificado:
    tiempo en acierto cache de 1 ciclo
    penalización por fallo de 100 ciclos

  - Tiempo medio de acceso:

    97% aciertos: 1 ciclo + 0.03 x 100 ciclos = **4 ciclos**

    99% aciertos: 1 ciclo + 0.01 x 100 ciclos = **2 ciclos**

- **Por eso se usa "tasa de fallo" en lugar de "tasa de acierto"**

¡Hora de juego!

Conectarse a:

https://swad.ugr.es > EC > Evaluación > Juegos

Memoria II: Cache

- Organización y Funcionamiento de la memoria cache
- **Impacto de la cache en el rendimiento**
  - Modelo de evaluación
  - La montaña de memoria
- Programación de código aprovechando la cache

Jerarquía de memoria

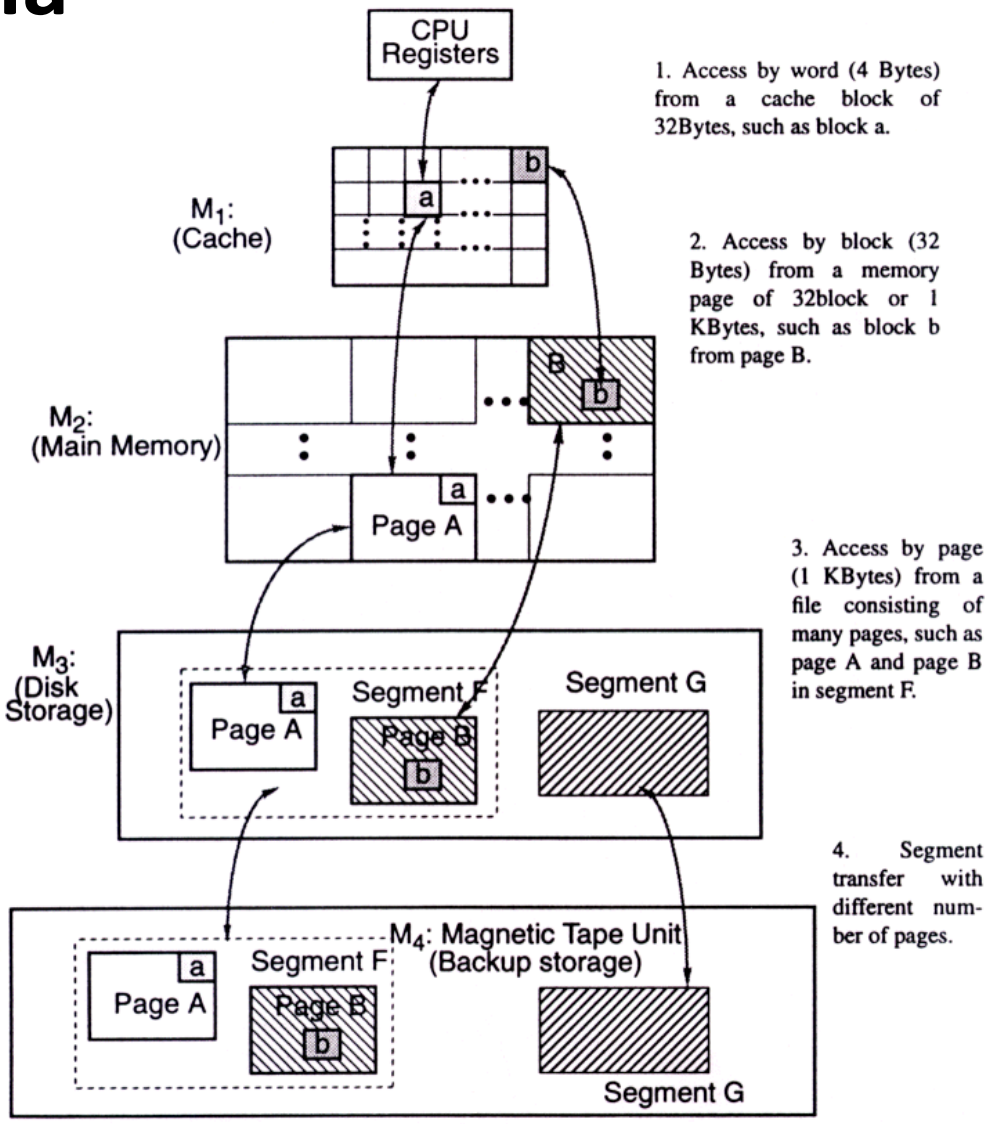
- **Parámetros que caracterizan cada nivel *i***
  - Los dispositivos de almacenamiento (registros, memorias, discos y unidades de cinta), se caracterizan por:
    - **Tiempo de acceso** (t_i):
      - Tiempo desde que se inicia una lectura hasta que llega la palabra deseada.
    - **Tamaño de la memoria** (s_i):
      - Número de bytes, palabras, sectores, etc., que se pueden almacenar en el dispositivo de memoria.
    - **Coste por bit o por byte** (c_i).
    - **Ancho de banda** (b_i):
      - Velocidad a la que se transfiere información desde un dispositivo.
    - **Unidad de transferencia** (x_i):
      - Tamaño de la unidad de información que se transfiere entre el nivel *i* y el *i*+1.

Se verifica que:

$$t_i < t_{i+1}$$
$$s_i < s_{i+1}$$
$$c_i > c_{i+1}$$
$$b_i > b_{i+1}$$
$$x_i < x_{i+1}$$

Jerarquía de memoria

- **Propiedad de inclusión**
  - $M_1 \subset M_2 \subset M_3 \subset \ldots \subset M_n$
  - Si una palabra se encuentra en $M_i \Rightarrow$ copias de esa palabra también se encuentran en M_{i+1}, M_{i+2}, ..., M_n.
  - Sin embargo, una palabra almacenada en M_{i+1} puede no estar en M_i, y si es así, tampoco estará en M_{i-1}, M_{i-2}, ..., M_1.



Modelo de evaluación de la jerarquía

- **Modelo de evaluación del rendimiento de una jerarquía de memoria (C. K. Chow, 1974*)**
  - Se asume que caches son inclusivas ($\sum_{i=1}^{j} a_i = A_j$)
    - y que compilador no reutiliza registros ($A_0 = 0$)
  - **Tasa de aciertos** A_i (*hit ratio*)
    - Porcentaje de información buscada que está en el nivel *i*.
    - En una jerarquía con *n* niveles:
      - $A_0 = 0$
      - $A_n = 1$
    - A_i depende de:
      - Capacidad del nivel *i* (s_i).
      - Granularidad de la transferencia de información.
      - Estrategia de administración de memoria.

** C. K. Chow. "On optimization of storage hierarchies." IBM Journal of Research and Development, pp. 194-203, May 1974.*

Modelo de evaluación de la jerarquía

- **Tasa de fallos F_i:** (*miss ratio*)
  - $F_i = 1 - A_i$, i=0, ..., n.
    - $F_0 = 1$
    - $F_n = 0$
- **Tasa de aciertos *específica* del nivel *i* a_i:**
  - Frecuencia de accesos con *primer* éxito al nivel i
  - Probabilidad de acceder con éxito a una información en nivel *i* y que esa información no se encuentre en los niveles 0 a *i*-1.

$$\sum_{i=1}^{n} a_i = 1$$

  - Dado que la información de M_j está también en M_k, $k > j$:
    - $a_i = A_i - A_{i-1}$, $i = 1, ..., n$
    - $a_1 = A_1$
    - $a_n = 1 - A_{n-1}$

Modelo de evaluación de la jerarquía

- Los objetivos al diseñar una memoria con *n* niveles son:
  ①Obtener un rendimiento cercano al de la memoria M_1 (la más rápida).
  ②Obtener un coste por bit cercano al de la memoria M_n (la más barata).

①Rendimiento:

- Cuantificable con el **tiempo medio de acceso a la jerarquía ($\overline{T}$)**.

Tiempo de acceso *efectivo* al nivel *i*-ésimo de la jerarquía (T_i):

$$T_i = \sum_{j=1}^{i} t_j$$

t_j = tiempo de acceso *específico* del nivel *j*

$$\overline{T} = \sum_{i=1}^{n} a_i T_i$$

Sumatoria de la tasa de acierto *específica* de M_i multiplicada por el tiempo de acceso *efectivo* a M_i

Modelo de evaluación de la jerarquía

$$\overline{T} = \sum_{i=1}^{n} a_i \, T_i \qquad a_i = A_i - A_{i-1}$$

- **Sustituyendo a_i y T_i:**

$$\overline{T} = \sum_{i=1}^{n} \left[\left(A_i - A_{i-1} \right) \sum_{j=1}^{i} t_j \right] = \left(A_1 - A_0\right)t_1 + \left(A_2 - A_1\right)\left(t_1 + t_2\right) + \left(A_3 - A_2\right)\left(t_1 + t_2 + t_3\right) +$$

$$+ \ldots + \left(A_n - A_{n-1}\right)\left(t_1 + t_2 + t_3 + \ldots + t_n\right) =$$

$$= \left(A_1 - A_0 + A_2 - A_1 + A_3 - A_2 + \ldots + A_n - A_{n-1}\right)t_1 +$$

$$+ \left(A_2 - A_1 + A_3 - A_2 + \ldots + A_n - A_{n-1}\right)t_2 +$$

$$+ \left(A_3 - A_2 + \ldots + A_n - A_{n-1}\right)t_3 +$$

$$+ \ldots + \left(A_n - A_{n-1}\right)t_n =$$

$$= \sum_{i=1}^{n} \left(A_n - A_{i-1}\right)t_i = \sum_{i=1}^{n} \left(1 - A_{i-1}\right)t_i =$$

$$= \sum_{i=1}^{n} F_{i-1} t_i$$

Modelo de evaluación de la jerarquía

- **②Coste por bit:**
  - El coste por bit promedio $c(n)$ de un sistema de n niveles es:

$$c(n) = \frac{\sum_{i=1}^{n} c_i s_i}{\sum_{i=1}^{n} s_i} + c_0$$

coste total

coste de interconexión entre niveles

tamaño total

Cache: modelo de evaluación

- **Modelo aplicado a un sistema de solo 2 niveles (memoria cache y memoria principal).**
  - $t_1 = t_c$: tiempo de acceso de la memoria cache (*específico*).
  - $a_1 = A_1 = A$: tasa de acierto (*hit ratio*) de la memoria cache.
  - $t_2 = t_m$: tiempo de acceso a la memoria principal (*específico*).
  - Tiempo medio de acceso:

$$\overline{T} = At_c + (1-A)(t_c + t_m)$$

Acierto ⇒ no se accede a MP

Fallo ⇒ se accede a caché y a MP

  - γ: razón entre los tiempos de acceso a la MP y a la cache

$$\gamma = \frac{t_m}{t_c}$$

Caché: modelo de evaluación

- Eficiencia de un sistema que emplea memoria cache:

$$E = \frac{t_c}{\overline{T}}, \quad 0 < E \leq 1$$

$$E = \frac{t_c}{At_c + (1-A)(t_c + t_m)} = \frac{1}{A + (1-A)\left(1 + \frac{t_m}{t_c}\right)} =$$

$$= \frac{1}{A + (1-A)(1+\gamma)} = \frac{1}{A + 1 - A + \gamma(1-A)} = \frac{1}{1 + \gamma(1-A)}$$

  - E máxima cuando A=1 (todas las referencias en cache).
  - $\gamma\uparrow \Rightarrow E\downarrow$; $A\downarrow \Rightarrow E\downarrow$. Interesa γ baja y A alta.
  - Ejemplo:
    - t_c = 15 ns
    - t_m = 100 ns
    - A = 0,9

$$\overline{T} = At_c + (1-A)(t_c + t_m) = 0{,}9 \cdot 15 + 0{,}1 \cdot 115 = 25\text{ns}$$

$$\gamma = \frac{t_m}{t_c} = \frac{100}{15} = 6{,}\widehat{6} \quad E = \frac{1}{1 + \gamma(1-A)} = \frac{1}{1 + 6{,}\widehat{6} \cdot 0{,}1} = 0{,}6$$

Caché separada / unificada

- **Caches separadas de datos / instrucciones**
  - Es común particionar la caché en dos módulos diferentes:
    - **Caché de datos.**
      - La localidad de los datos no es tan buena como la de las instrucciones ⇒ la caché de datos es **menos eficiente**.
      - Es **más compleja** por la posibilidad de **modificación** de los datos.

      ⇒ Muchos sistemas no admiten caché de datos.
    - **Caché de instrucciones.**
      - Es fácil que bucles y pequeñas rutinas entren totalmente en caché, permitiendo su ejecución sin necesidad de acceder a MP.
      - Se simplifica la caché, ya que es habitual que se prohíba escribir en las localizaciones donde se encuentran las instrucciones ⇒ caché de **sólo lectura**.
  - ✓ Se pueden emitir direcciones de instrucción y dato a la vez, doblando el ancho de banda entre caché y procesador.
  - ✓ Se puede optimizar cada caché por separado:
    - Diferentes capacidades, tamaños de bloque, asociatividades, etc.

Caché separada / unificada

Las caches de instrucciones tienen menor frecuencia de fallos que las de datos.

Tamaño	Instrucción sólo	Sólo datos	Unificada
0,25 KB	22,2 %	26,8 %	28,6 %
0,50 KB	17,9 %	20,9 %	23,9 %
1 KB	14,3 %	16,0 %	19,0 %
2 KB	11,6 %	11,8 %	14,9 %
4 KB	8,6 %	8,7 %	11,2 %
8 KB	5,8 %	6,8 %	8,3 %
16 KB	3,6 %	5,3 %	5,9 %
32 KB	2,2 %	4,0 %	4,3 %
64 KB	1,4 %	2,8 %	2,9 %
128 KB	1,0 %	2,1 %	1,9 %
256 KB	0,9 %	1,9 %	1,6 %

Frecuencia de fallos para caches de distintos tamaños de sólo datos, sólo instrucciones, y unificadas. Los datos son para una cache asociativa de 2 vías utilizando reemplazo LRU con bloques de 16 bytes para un promedio de trazas de usuario/sistema en la VAX-11 y trazas de sistema en el IBM 370 [Hill 1987]. El porcentaje de referencias a instrucciones en estas trazas es aproximadamente del 53 por 100.

¿Cuál tiene una frecuencia de fallos menos: una caché de instrucciones de 16 KB + una caché de datos de 16 KB o una caché unificada de 32 KB?

Frecuencia de fallos para la caché particionada:

53%·3,6%+47%·5,3% = 4,4%

Una caché unificada de 32 KB tiene una frecuencia de fallos similar (4,3%).

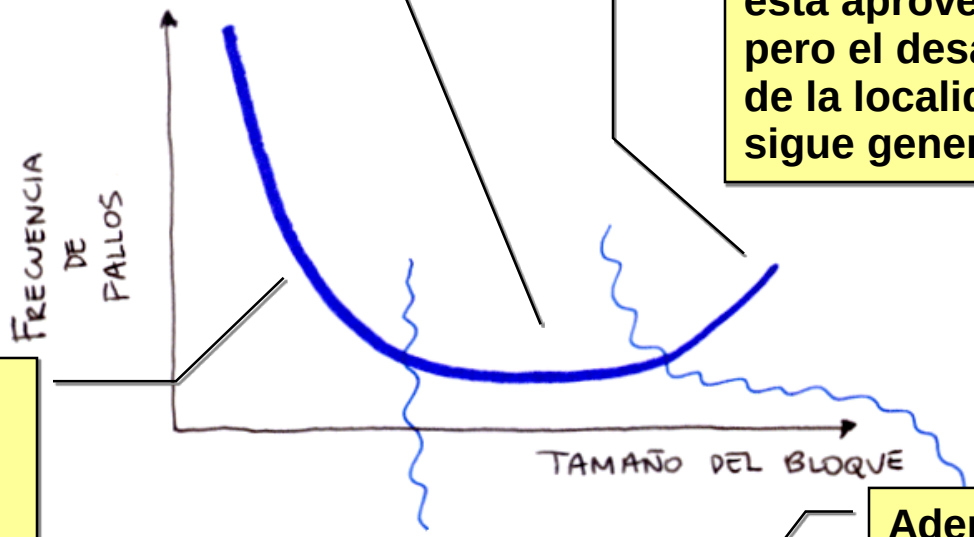
(Ver, sin embargo, las ventajas de la transparencia anterior).

Caché: tamaño

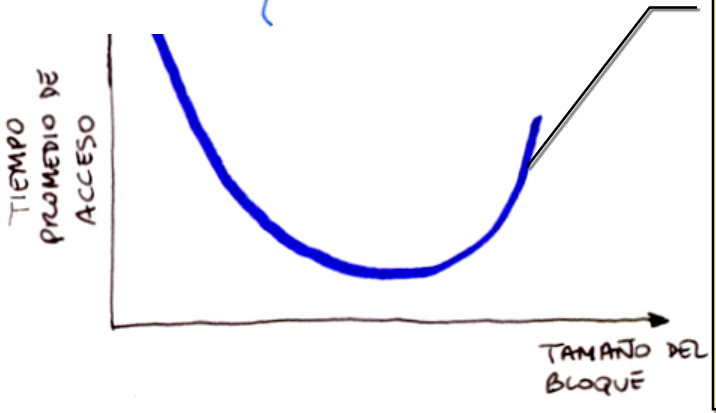
- **Tamaños del bloque y de la caché**
  - Para un tamaño de caché fijo:

Ambos efectos de localidad se compensan.

Aquí la localidad espacial ya está aprovechada al máximo, pero el desaprovechamiento de la localidad temporal sigue generando faltas.



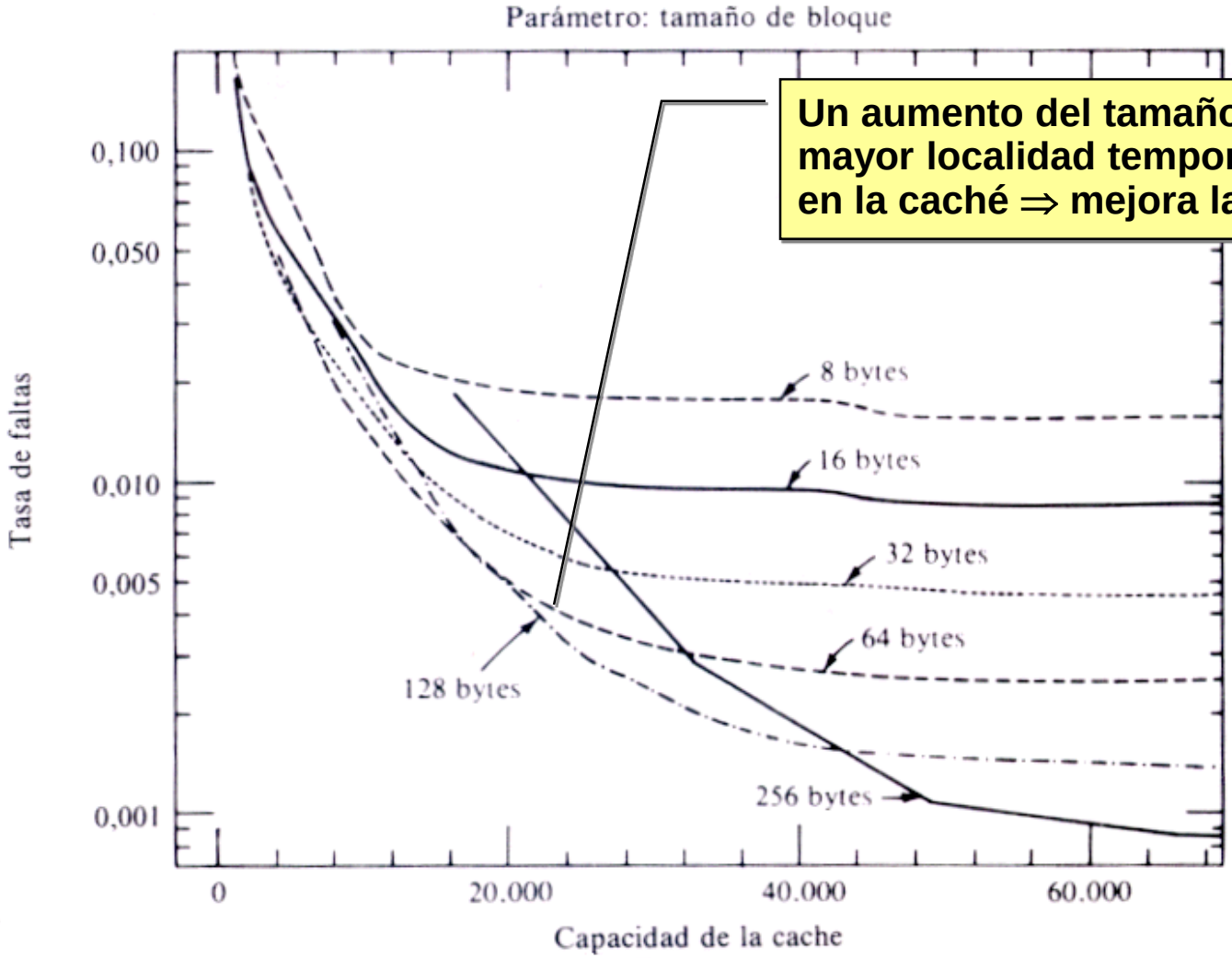
La tasa de faltas disminuye al aumentar el tamaño del bloque, pues se aprovecha mejor la localidad espacial. Pero cuando tamaño del bloque ↑ ⇒ n° de bloques ↓



Además: tamaño de bloque ↑↑ ⇒ tiempo de transferencia ↑ ⇒ tiempo promedio de acceso ↑ según el producto frecuencia de fallos × tiempo de penalización por fallos.

Caché: tamaño

- Para un tamaño de bloque fijo:



Memoria II: Cache

- Organización y Funcionamiento de la memoria cache
- Impacto de la cache en el rendimiento
  - Modelo de evaluación
  - La montaña de memoria
- Programación de código aprovechando la cache

La montaña de memoria

- **Rendimiento de lectura (ancho de banda de lectura)**
  - Número de bytes leídos de memoria por segundo (MB / s)

- **Montaña de memoria: rendimiento de lectura medido en función de la localidad espacial y temporal.**
  - Manera compacta de caracterizar el rendimiento del sistema de memoria.

Función test para montaña memoria

```
long data[MAXELEMS];  /* Array global a recorrer */

/* test – Iterar sobre los primeros "elems" elementos
 *        del array "data" con paso de "stride",
 *        usando desenrollado de bucles x4
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combinar 4 elementos a la vez */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Terminar con los elementos restantes */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Llamar a `test()` con muchas combinaciones de `elems` y `stride`

Para cada `elems` y `stride`:

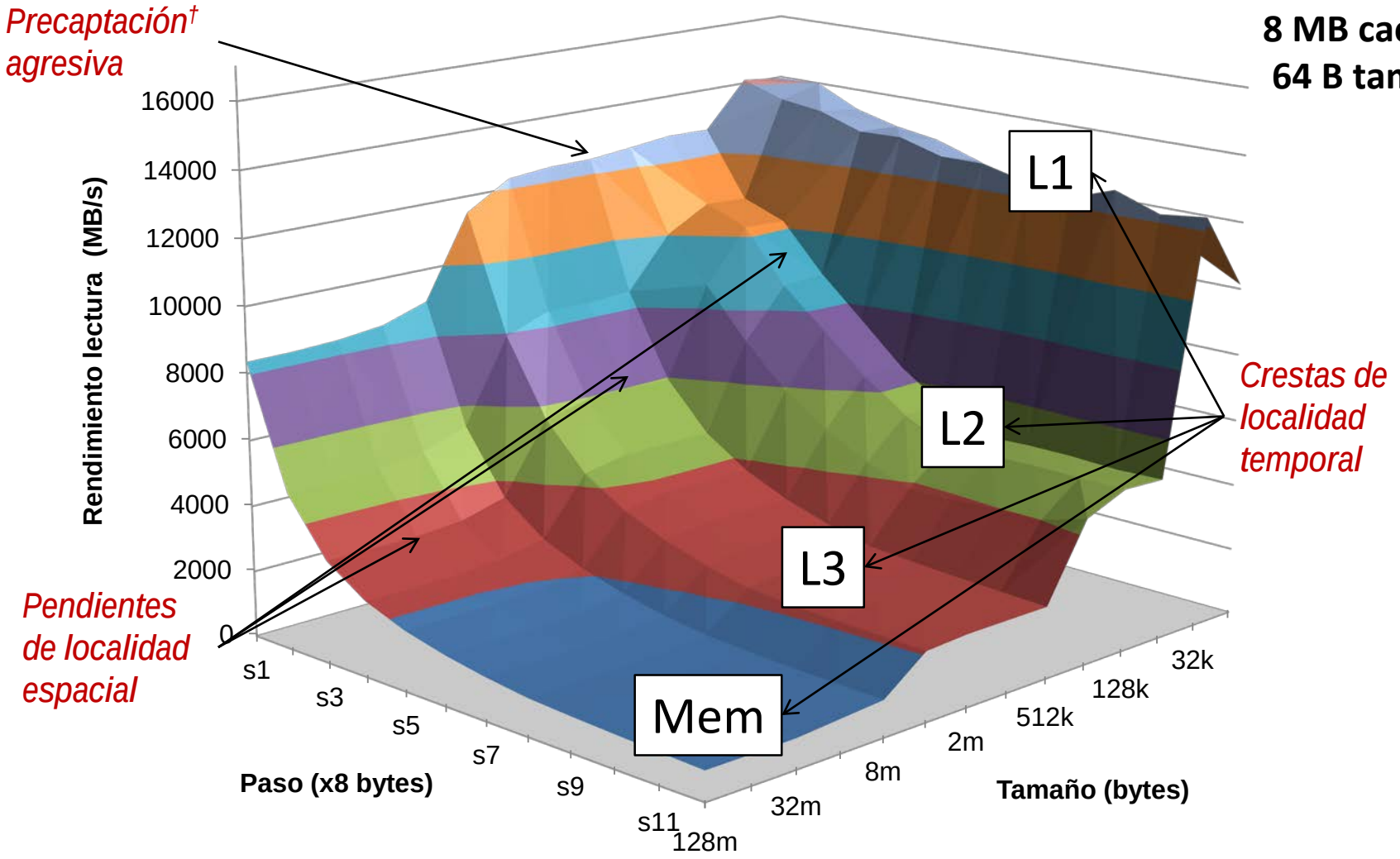
1. Llamar una vez a `test()` para calentar las caches

2. Llamar a `test()` otra vez y medir el rendimiento de lectura (MB/s)

† *for (...; i<=limit; ...)*

La Montaña de Memoria

Core i7 Haswell
2.1 GHz
32 KB cache-d L1
256 KB cache-d L2
8 MB cache L3
64 B tam. bloque



Memoria II: Cache

- Organización y Funcionamiento de la memoria cache
- Impacto de la cache en el rendimiento
  - La montaña de memoria
- **Programación de código aprovechando la cache**

Para escribir código amigable con cache

- **Hacer que el caso más común vaya rápido**
  - Centrarse en los bucles internos de las funciones básicas

- **Minimizar los fallos en los bucles internos**
  - Referencias repetidas a variables son buenas (localidad temporal)
  - Patrones de referencia de paso-1 son buenos (localidad espacial)

Idea clave: nuestra noción cualitativa de localidad se cuantifica a través de nuestra comprensión de las memorias caché

Ejemplo: *SimAquarium*†

- **Enunciado:**
  - Calcular "centro masas" (posición promedio) de 1024 algas (array 32x32)
  - Elementos del array son structs de 2 ints (2 x 4B)
    - Ignorar división final
    - Resto accesos registros

```
struct algae_position {
        int x;
        int y;
};

struct algae_position_grid[32][32];
int total_x = 0, total_y = 0;
int i, j;
```

CS:APP 3rd Ed. p.673

El resto de variables se mantienen en registros

- **Parámetros cache**
  - tamaño 2KB
  - bloques 32B
  - correspondencia directa

† CS:APP 3rd Ed. Prob.6.18, pg.673-675

Ejemplo: *SimAquarium*

- **Cálculos:**
  - struct: 8B (2 ints)
  - bloque: 4 structs (8B x 4 = 32B)
  - fila: 32 struct = 8 bloques = 256B
  - cache: $2^{11} \div 2^{5} = 64$ bloques = 8 filas = 2KB
  - array: 32 filas (4x cache), 256 bloques, 1K structs, 8KB
- **Parámetros cache**
  - tamaño 2KB
  - bloques 32B
  - correspondencia directa

```
struct algae_position {
        int x;
        int y;
};

struct algae_position_grid[32][32];
int total_x = 0, total_y = 0;
int i, j;
```

CS:APP 3rd Ed. p.673

Disposición en memoria de arrays C (repaso)

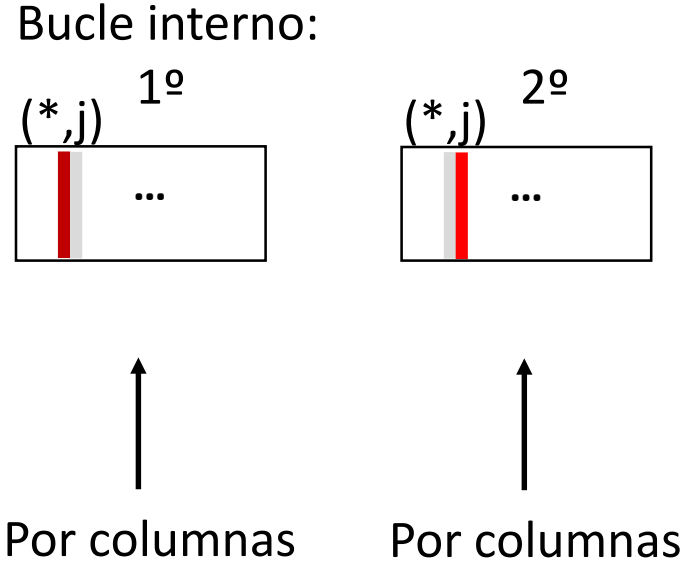
- **arrays C almacenados por filas *(row-major order)***
  - columnas en posiciones de memoria contiguas forman una fila
  - filas también contiguas, sin huecos entre ellas
- **Avanzar por las columnas de una fila:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - accede a elementos sucesivos
  - aprovecha localidad espacial si tamaño bloque B > sizeof(a_{ij}) bytes,
    - tasa de fallo = sizeof(a_{ij}) / B
- **Avanzar por las filas de una columna:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[i][0];
    ```
  - accede a elementos distantes → ¡sin localidad espacial!
    - tasa de fallo = 1 (100%) en cuanto #cols N ≥ B / sizeof(a_{ij})

      N x sizeof(a_{ij}) / B ≥ 1

Versión 1

```
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_x += grid[i][j].x;
  }
}
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674



Fallos por iteración del bucle interno:

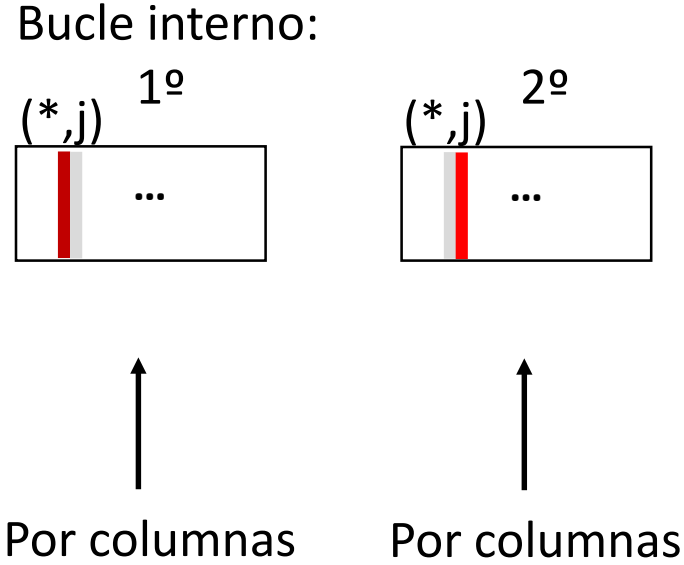
1º 2º

Tamaño bloque = 32B (4 structs)

Versión 1

```
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_x += grid[i][j].x;
  }
}
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674



Fallos por iteración del bucle interno:

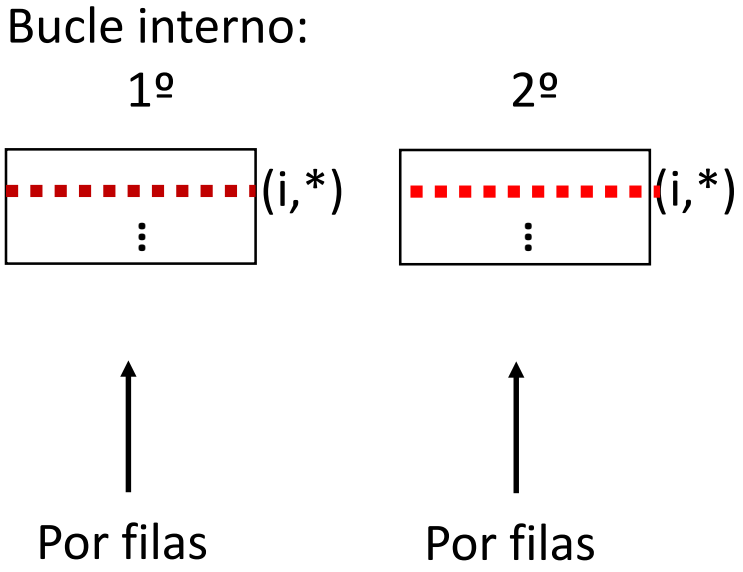
1º	2º
1	1

Tamaño fila (32 struct) >>
Tamaño bloque = 32B (4 structs)

Versión 2

```
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_x += grid[i][j].x;
  }
}
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674



Fallos por iteración del bucle interno:

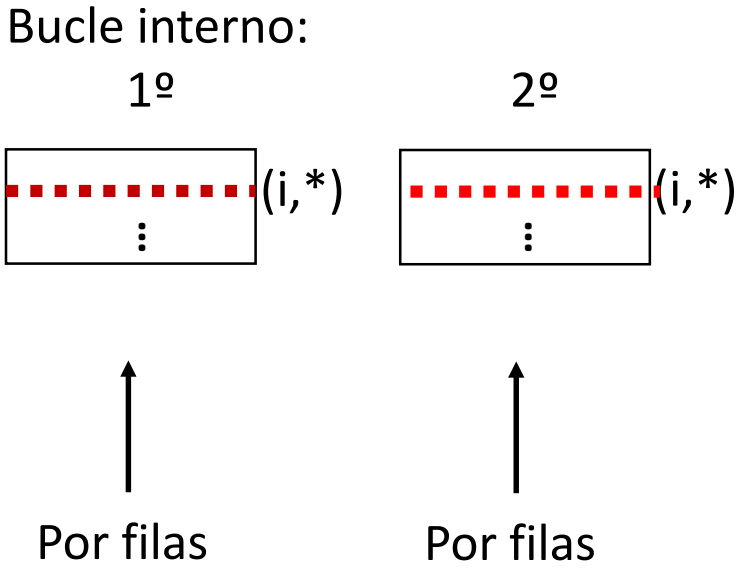
1º 2º

Tamaño bloque = 32B (4 structs)

Versión 2

```
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_x += grid[i][j].x;
  }
}
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674



Fallos por iteración del bucle interno:

1º	2º
0.25	0.25

Tamaño struct (8B) x 4 =

Tamaño bloque = 32B (4 structs)

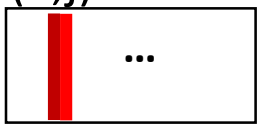
Versión 3

```
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674

Bucle interno:

(*,j)



Por columnas

Fallos por iteración del bucle interno:

1º

Tamaño bloque = 32B (4 structs)

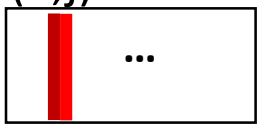
Versión 3

```
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674

Bucle interno:

(*,j)



Por columnas

Fallos por iteración del bucle interno:

1º

0.50

8B (str.) < 32B (blq.) < 256B (fila)

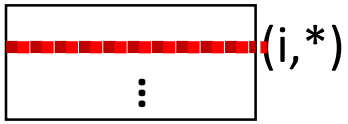
Tamaño bloque = 32B (4 structs)

Versión 4

```
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674

Bucle interno:



Por filas

Fallos por iteración del bucle interno:

1º

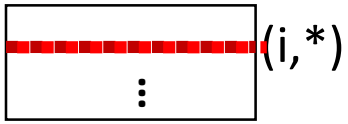
Tamaño bloque = 32B (4 structs)

Versión 4

```
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
}
```

CS:APP 3rd Ed. p.674

Bucle interno:



Por filas

Fallos por iteración del bucle interno:

1º

0.125

4B (int) < 32B (blq.)

Tamaño bloque = 32B (8 int)

SymAquarium: Resumen

```
for (j=0; j<32; j++)
  for (i=0; i<32; i++)
    total_x += grid[i][j].x;
for (j=0; j<32; j++)  {
  for (i=0; i<32; i++) {
    total_y += grid[i][j].y;
```

for$_{ji}$ x, for$_{ji}$ y:
- desaprovecha stride-1
- desapr. loc.espacial x-y
- promedio fallos/iter = **1**

```
for (i=0; i<32; i++)
  for (j=0; j<32; j++)
    total_x += grid[i][j].x;
for (i=0; i<32; i++)  {
  for (j=0; j<32; j++) {
    total_y += grid[i][j].y;
```

for$_{ij}$ x, for$_{ij}$ y:
- desapr. loc.espacial x-y
- promedio fallos/iter = **0.25**

```
for (j=0; j<32; j++)
  for (i=0; i<32; i++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

for$_{ji}$ x, y:
- desaprovecha stride-1
- promedio fallos/iter = **0.50**

```
for (j=0; j<32; j++)
  for (i=0; i<32; i++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

for$_{ij}$ x, y:
- aprovecha ambos
- promedio fallos/iter = **0.125**

Memoria II: Cache

- Organización y Funcionamiento de la memoria cache
- Impacto de la cache en el rendimiento
  - Modelo de evaluación
  - La montaña de memoria
- Programación de código aprovechando la cache

Resumen de cache

- **Las memorias cache pueden tener un impacto significativo en el rendimiento**

- **¡Puedes escribir tus programas para aprovechar esto!**
  - Céntrate en los bucles internos, donde se producen la mayor parte de los cálculos y de los accesos a memoria
  - Intenta maximizar la localidad espacial leyendo los datos secuencialmente con paso 1
  - Intenta maximizar la localidad temporal utilizando un dato con la mayor frecuencia posible una vez que se haya leído de memoria

Guía de trabajo autónomo (4h/s)

- **Estudio:** del Cap.6 CS:APP (Bryant/O'Hallaron)
  - Cache Memories (incl. ***fully associative caches***)
    - § 6.4 pp.650-669
  - Writing Cache-Friendly Code
    - § 6.5 pp.669-674
  - Impact of Caches on Program Performance
    - § 6.6 pp.675-684
- **Ejercicios:** del Cap.6 CS:APP (Bryant/O'Hallaron)
  - Probl. 6.9 § 6.4.1, p.652
  - Probl. 6.10 – 6.11 § 6.4.2, p.660
  - Probl. 6.12 – 6.16 § 6.4.4, pp.664-666
  - Probl. 6.17 – 6.20 § 6.5, pp.672-675
  - Probl. 6.21 § 6.6, p.679

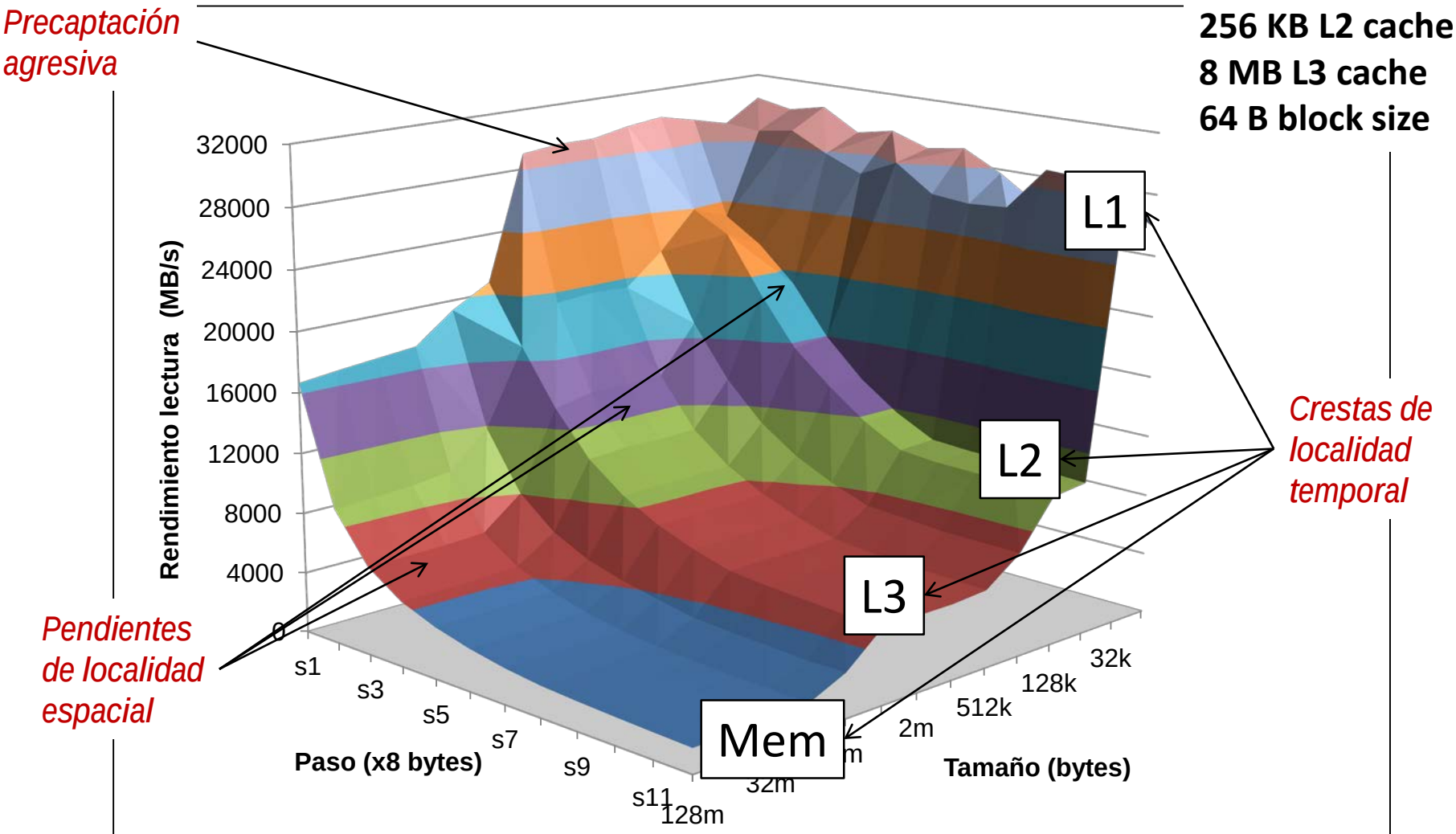
Bibliografía:

[BRY16] Cap.6 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/C.1 BRY com

Diapositivas suplementarias

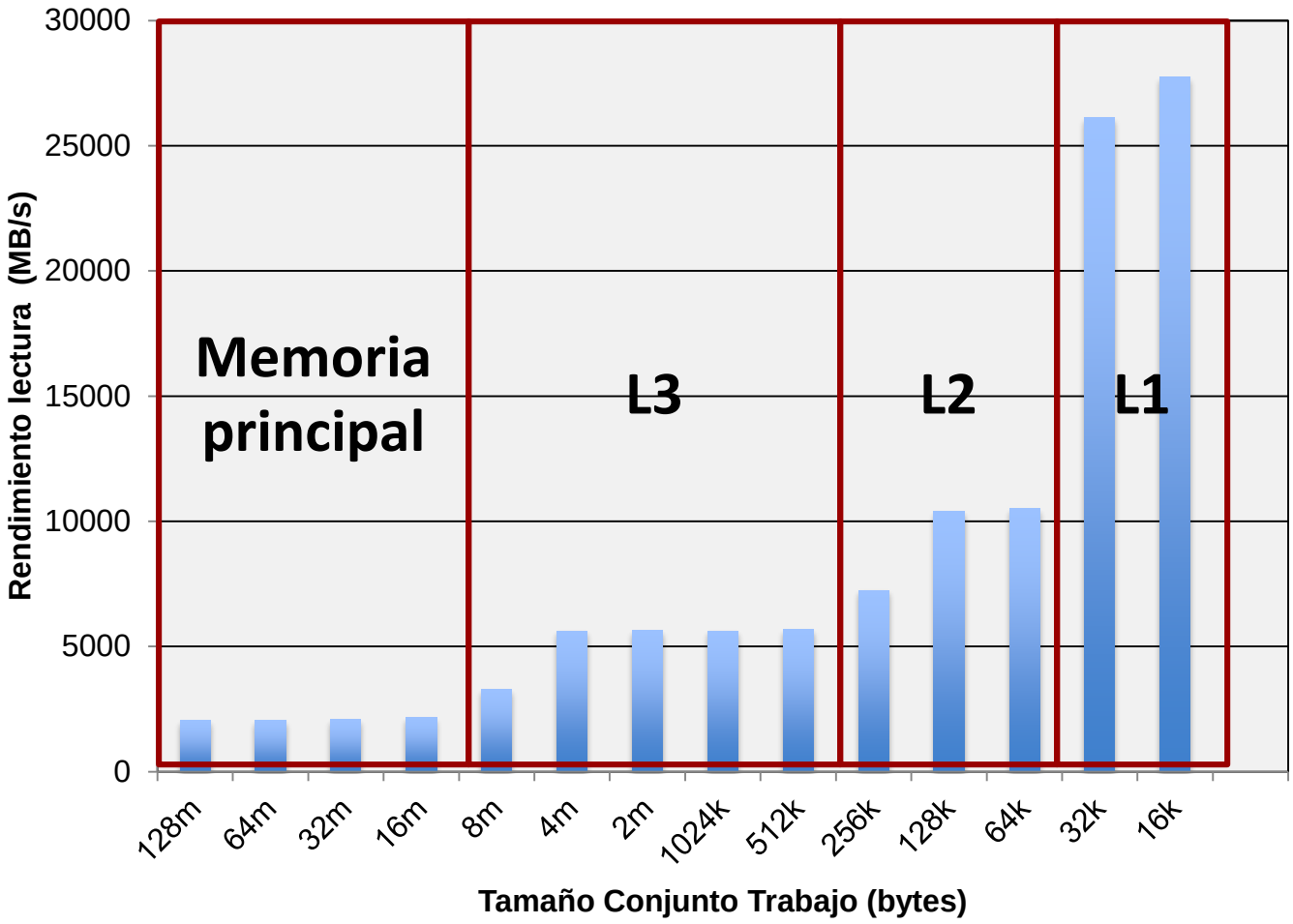
La Montaña de Memoria

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Efectos de la capacidad de cache
(tomado de la Montaña de Memoria)

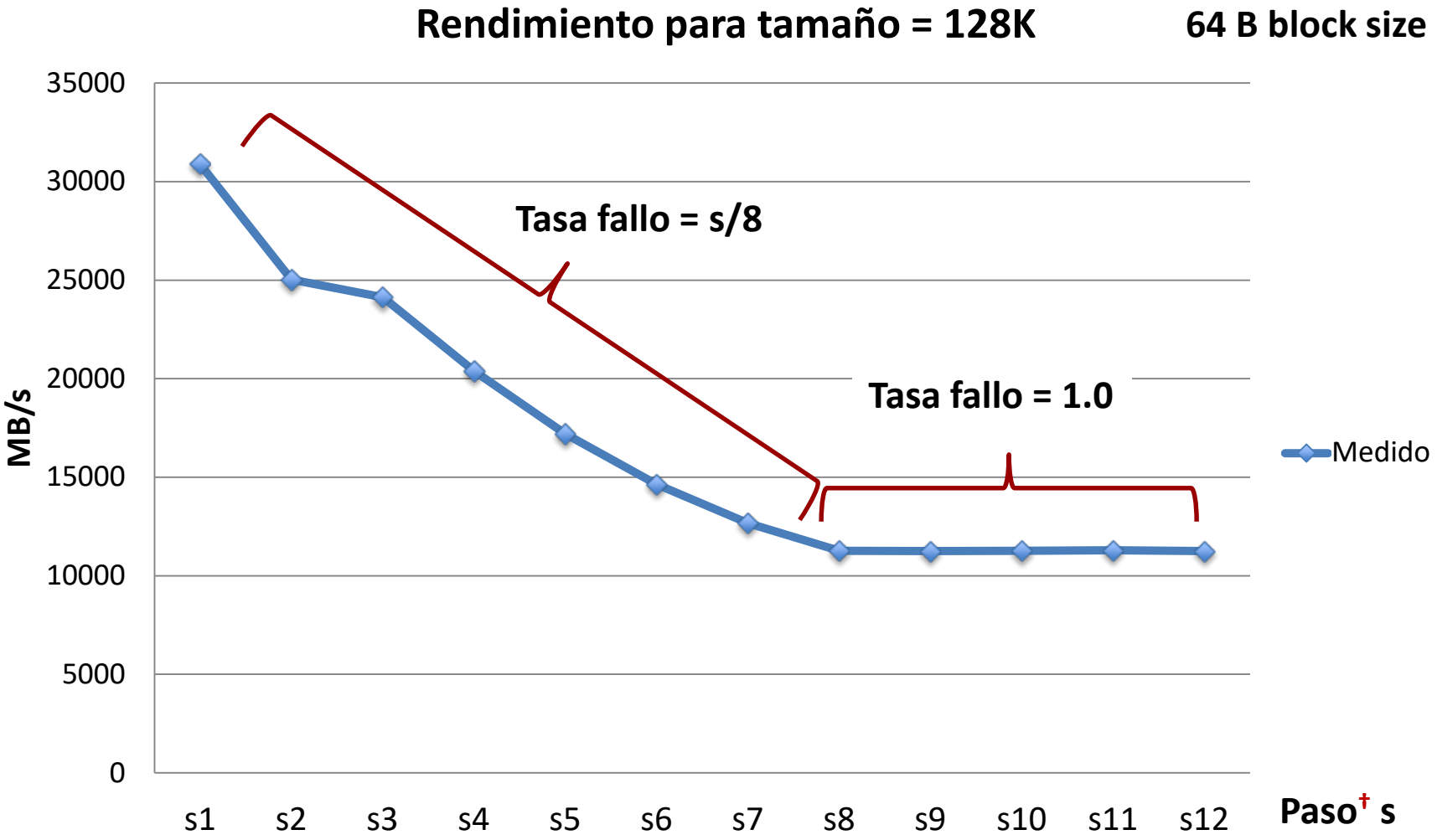
Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Corte transversal montaña memoria con paso=8

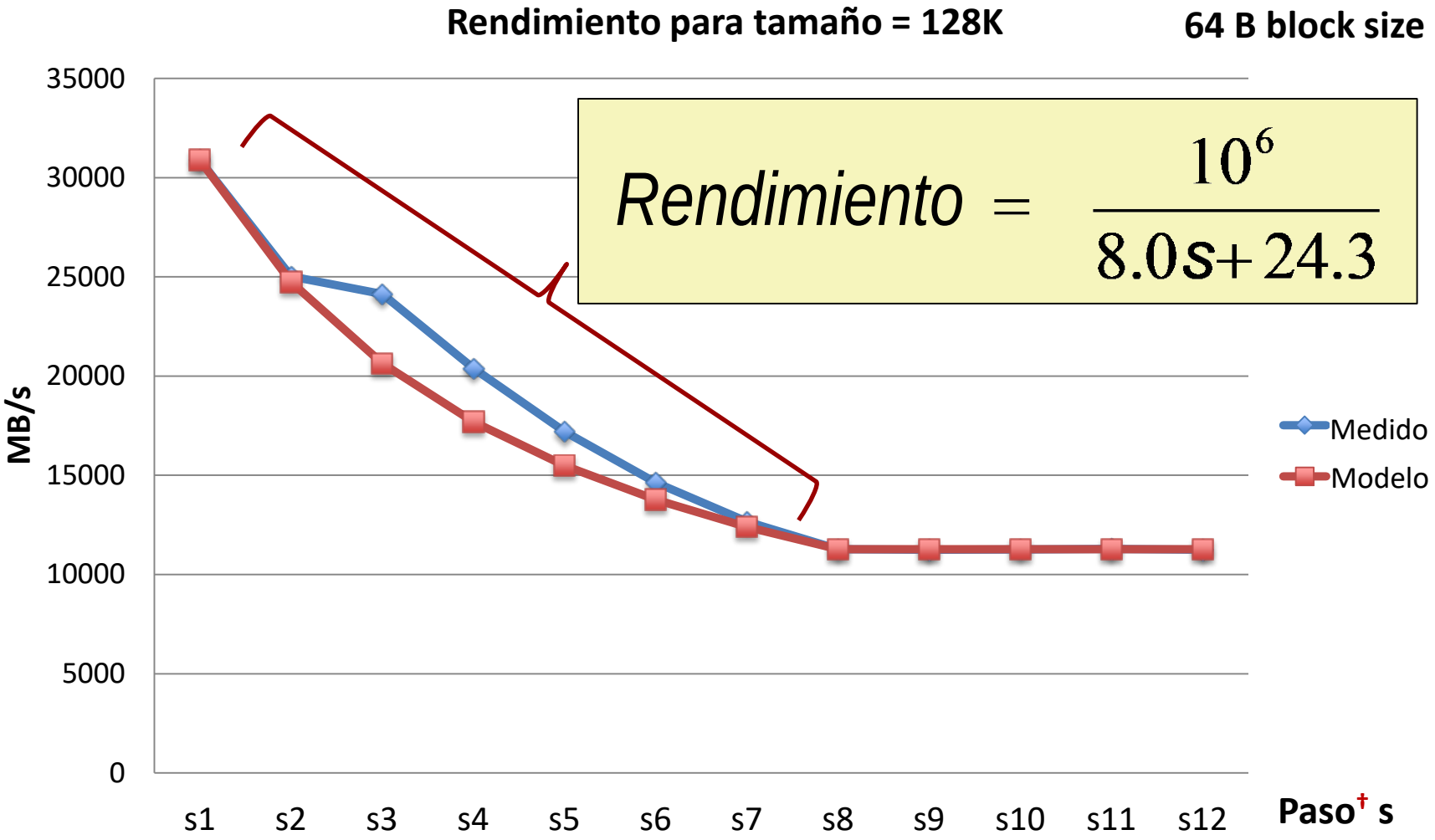
Efectos del tamaño bloque cache
(tomado de la Montaña de Memoria)

Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Modelando efectos tamaño bloque
(de la Montaña de Memoria)

Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Montaña Memoria 2008

Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size