Memorias cache

95.57/75.03 Organización del computador

Docentes: Patricio Moreno y Adeodato Simó

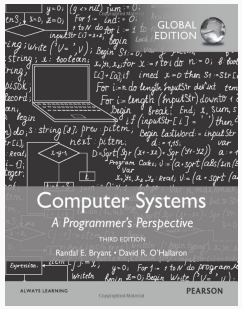
2.^{do} cuatrimestre de 2020

Última modificación: Tue Aug 4 14:36:17 2020 -0300

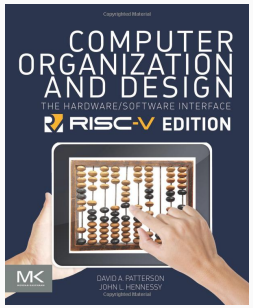
Facultad de Ingeniería (UBA)

Créditos

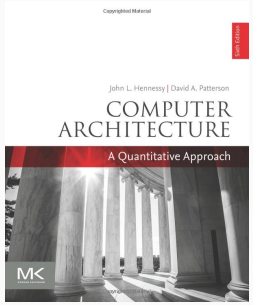
Para armar las presentaciones del curso nos basamos en:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.

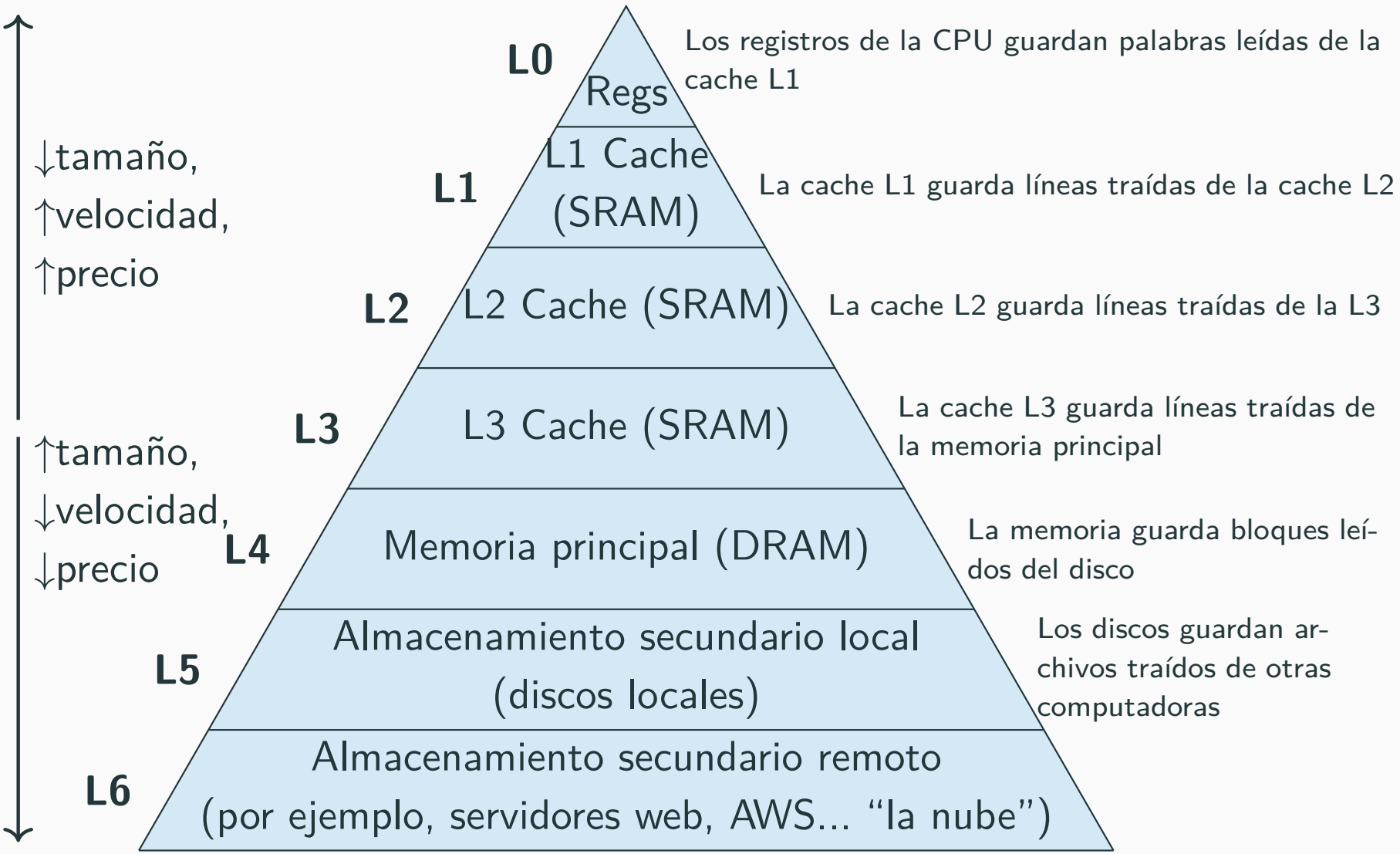


J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

Tabla de contenidos

Tabla de contenidos

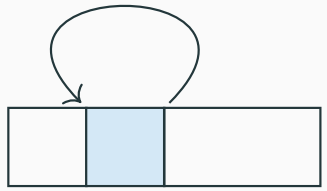
Jerarquía de memoria



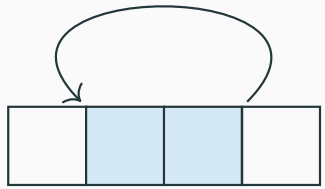
Principio de localidad

- Los programas tienden a acceder a una parte reducida de su espacio de memoria en un tiempo acotado; utilizan instrucciones o datos en direcciones **cercanas** o **iguales** a las usadas recientemente.

- Localidad temporal
  - Es probable que los items accedidos recientemente sean reutilizados
    - por ejemplo: instrucciones en un ciclo, variables
  - direcciones **iguales**



- Localidad espacial
  - Los items que se encuentran cerca suelen ser reutilizados
    - por ejemplo: acceso secuencial a instrucciones, datos en arreglos
  - direcciones **cercanas**



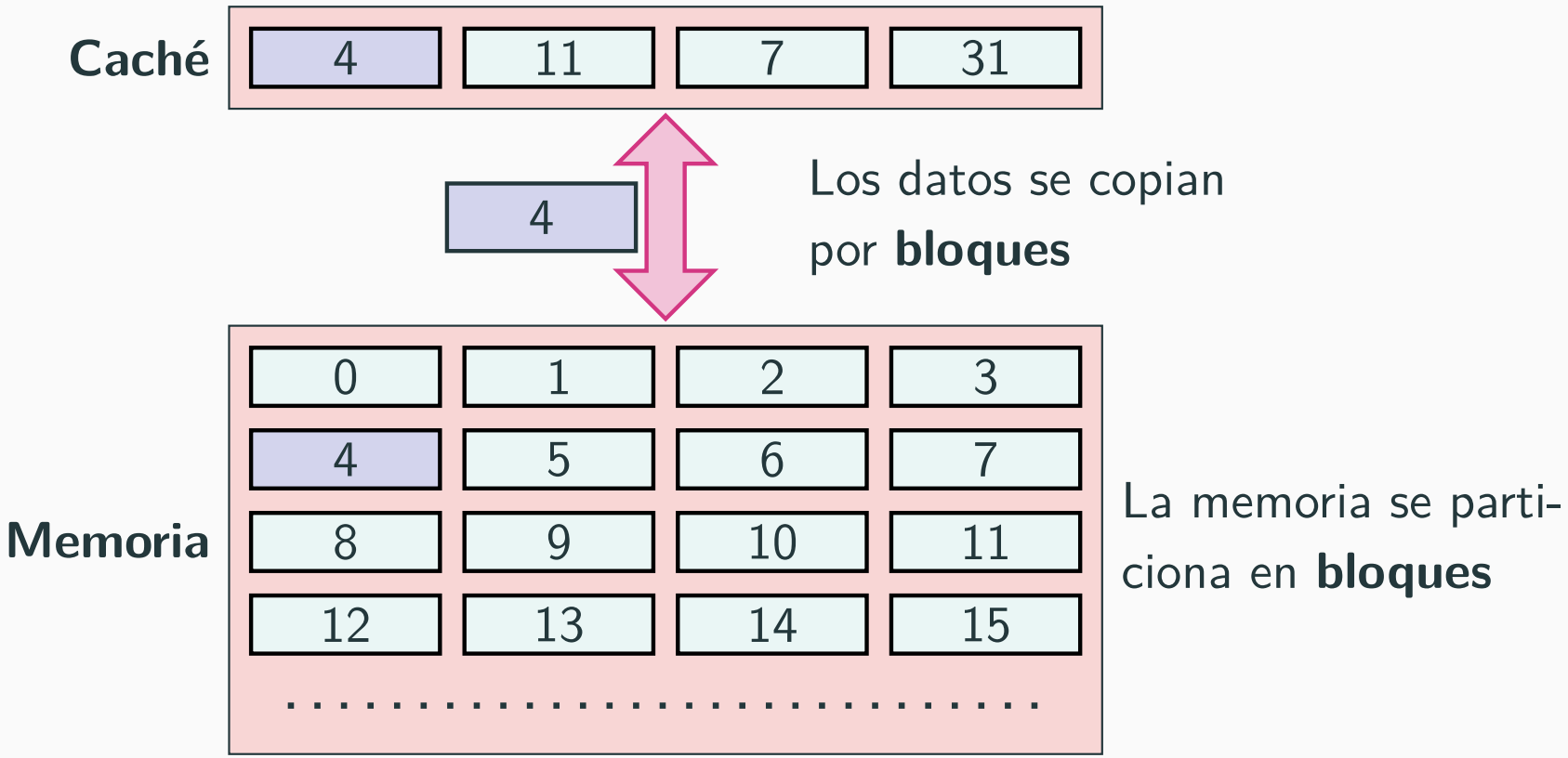
Caches

- **Cache**: es un dispositivo de almacenamiento de datos más rápido y de menor capacidad que actúa como *staging area* de un subconjunto de datos almacenados en un dispositivo más lento y de mayor capacidad
- Idea fundamental de la jerarquía de memorias
  - para cada k, el dispositivo de menor capacidad y mayor velocidad en el nivel k (L_k) sirve de cache para el dispositivo en el nivel $k+1$ (L_{k+1})
- ¿Por qué funciona la jerarquía de memorias?
  - por localidad, el software tiende a acceder con mayor frecuencia a los datos del nivel k que a los del nivel $k+1$.
- **Idealmente**: la jerarquía de memorias crea un *pool* de almacenamiento con el coste del almacenamiento en la base de la pirámide, y el tiempo de acceso del dispositivo en la cima de la misma.

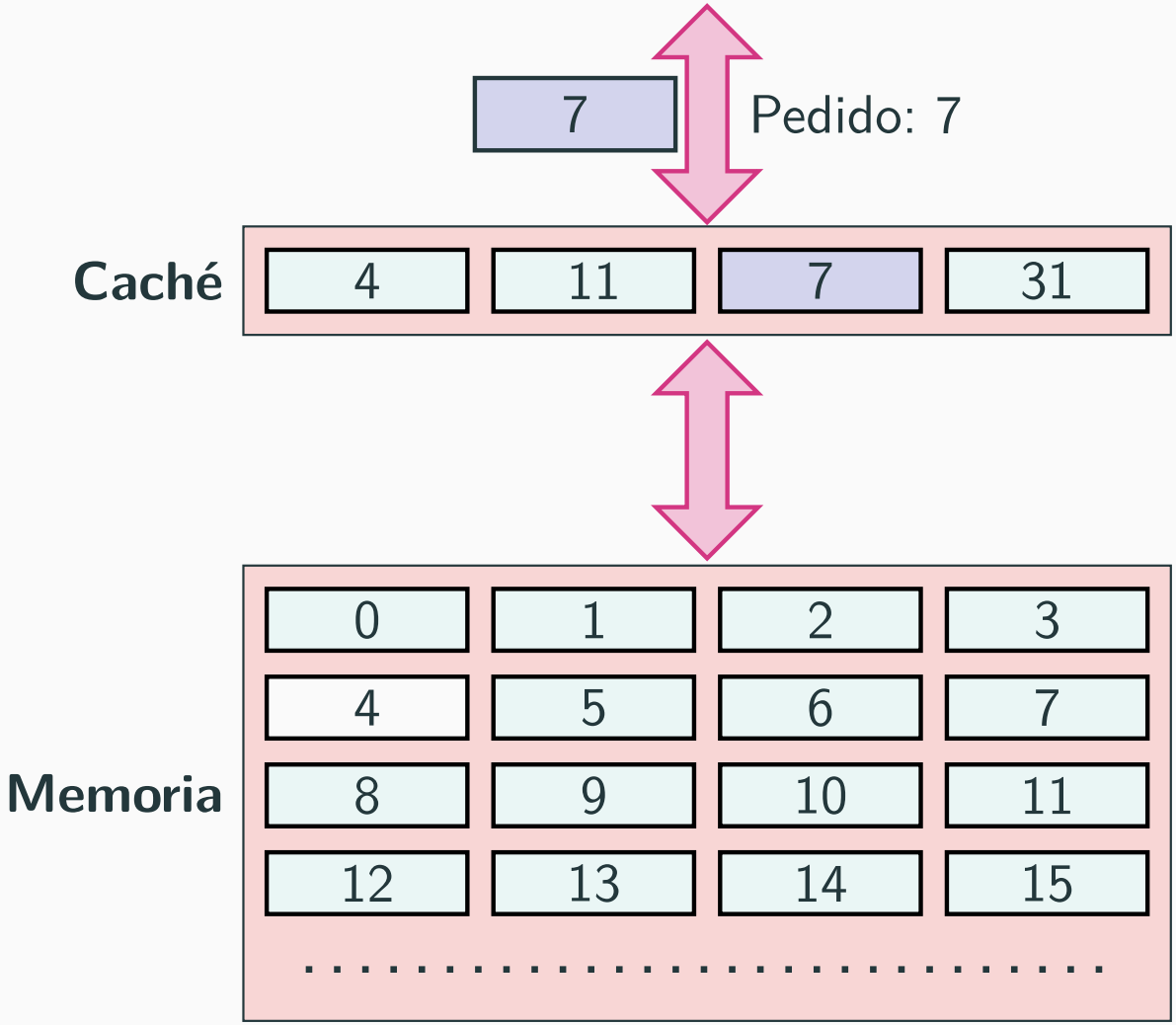
Definiciones

- Bloque o línea: unidad de copiado
  - Puede abarcar varias palabras
- *Hit*: el dato pedido está presente en el nivel superior
- *Miss*: el dato **no** se encuentra
  - el bloque se copia del nivel inferior
    - demora en el procedimiento: penalidad del *miss* (*miss penalty*)
  - Luego se pide el dato y habrá un *hit*
- *Hit ratio*: hits/accesos
- *Miss ratio*: misses/accesos = 1 - hit ratio

Definiciones



Definiciones: cache *hit*



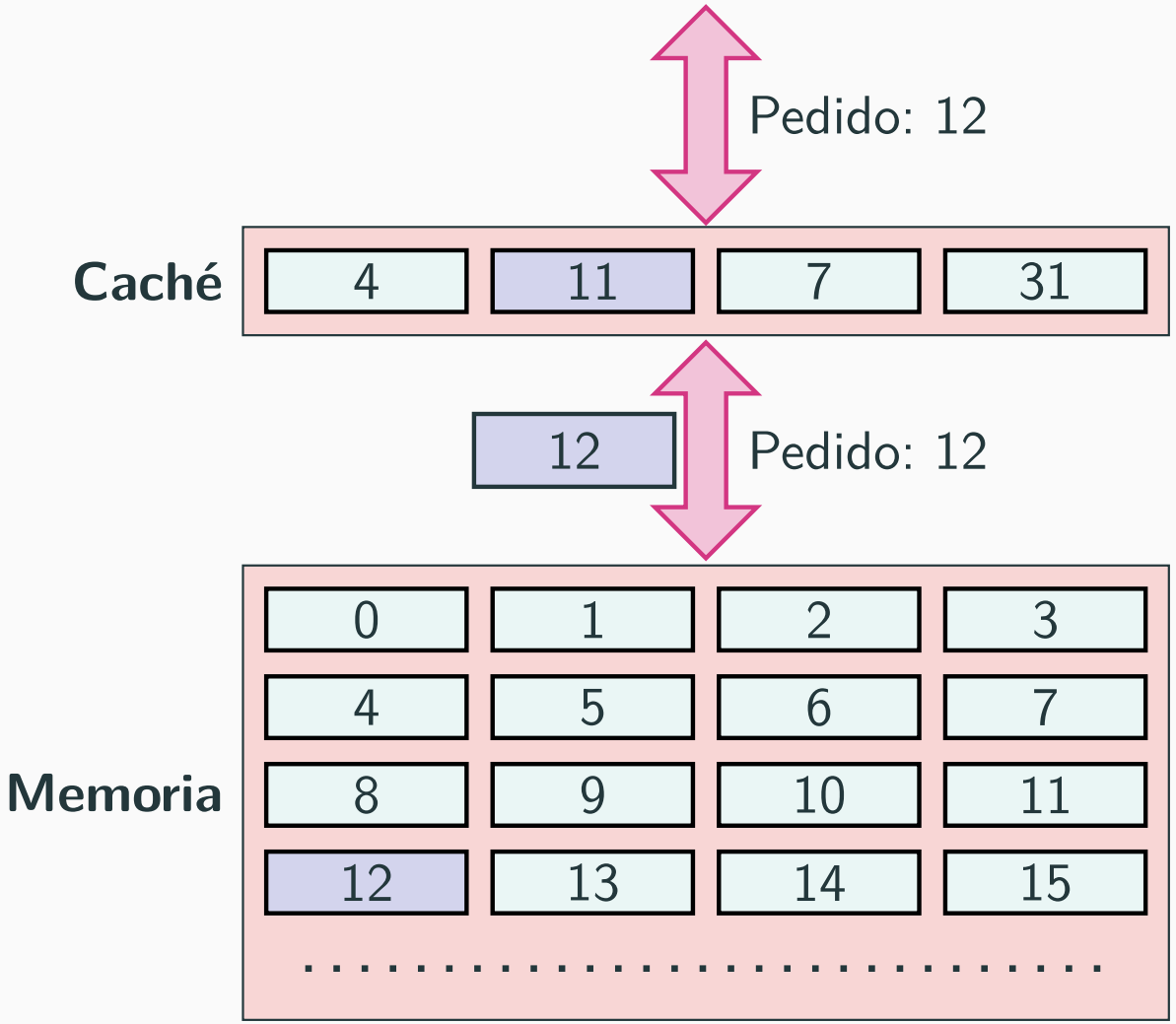
Se pide el dato que está en el **bloque** 7 de la memoria.

El dato **está** en la memoria caché.

Es un **hit**.

No se accede a la memoria.

Definiciones: cache *miss*



Se pide el dato que está en el **bloque** 12 de la memoria.

El dato **no** está en la memoria caché.

Es un **miss**.

Se le pide el bloque a la memoria.

Definiciones: almacenamiento en cache

Al almacenar un bloque en la memoria cache ocurren 2 cosas:

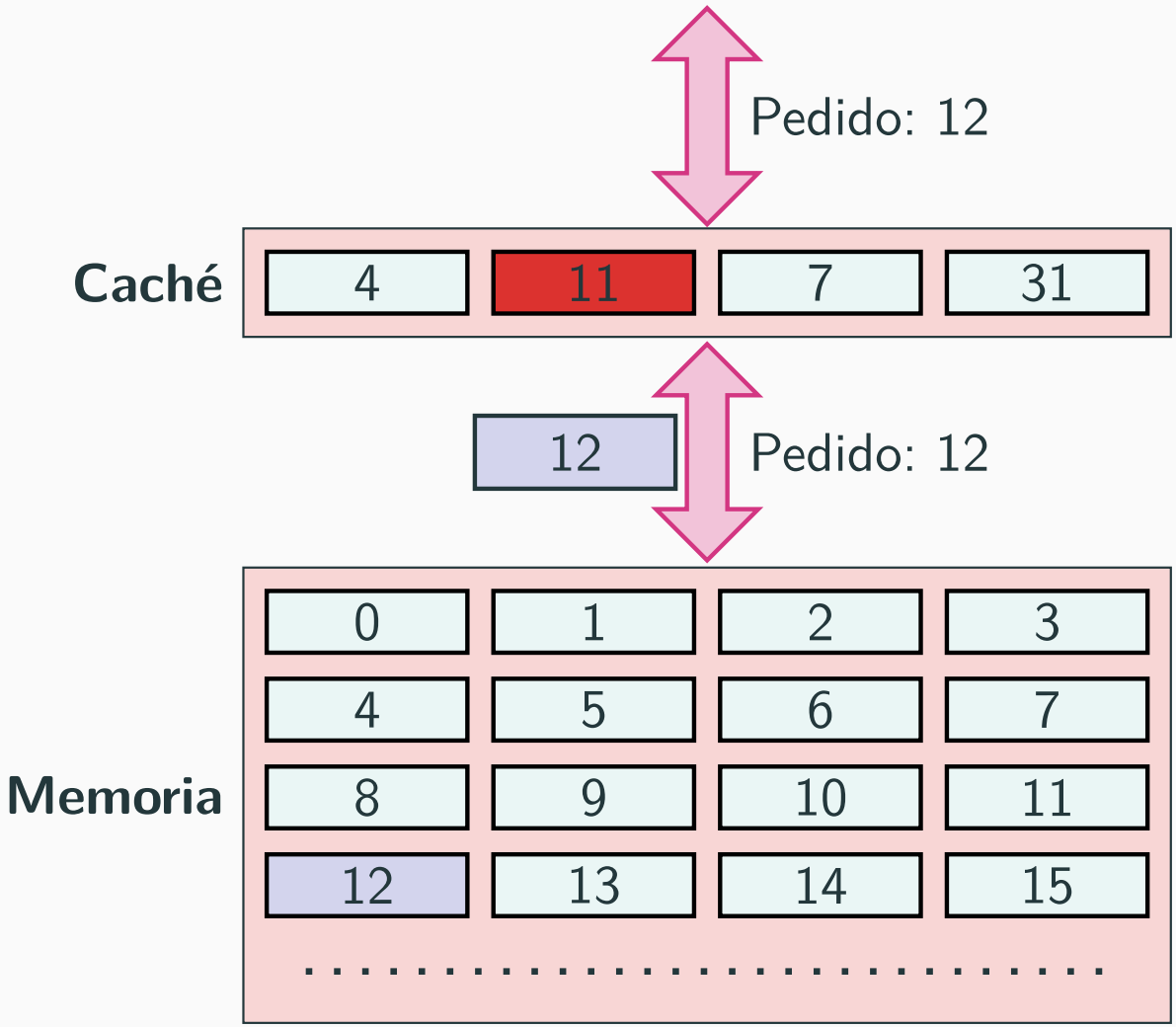
- el bloque se copia a la memoria cache: se lo *ubica*
- lo que estaba antes es reemplazado: se lo *reemplaza*

Para esto existen algoritmos, o políticas:

política de ubicación (en inglés: *placement policy*) determina dónde se ubica el bloque dentro de las opciones.

política de reemplazo (en inglés: *replacement/eviction policy*) determina qué bloque se reemplaza.

Definiciones: ubicación y reemplazo



El bloque 1 de la memoria caché, que contiene al bloque 11 de la memoria, es **reemplazado**.

El bloque 12 de la memoria es **ubicado** en el bloque 1 de la caché.

Políticas de ubicación y reemplazo

Tipos de fallos (*misses*) en la caché

Fallos en frío/forzosos/compulsivos

Ocurren porque la caché comienza *vacía* y se da la primera vez que se referencia un bloque.

Fallos por capacidad

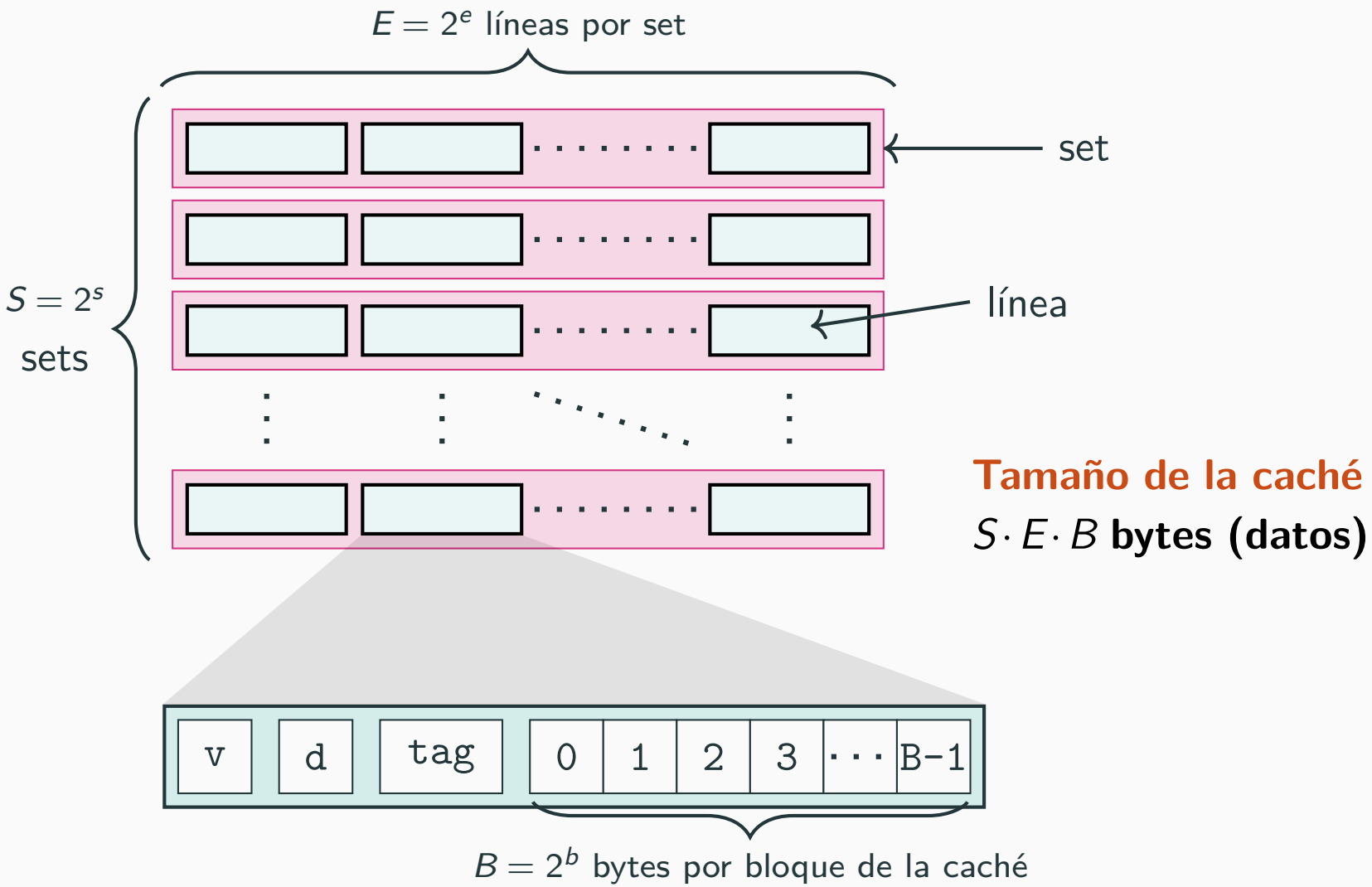
Ocurren cuando el conjunto de bloques activos en la caché (**conjunto de trabajo / *working set***) es mayor que la caché.

Fallos por conflictos

Ocurren cuando, siendo la caché lo suficientemente grande, más de un bloque se quiere ubicar en la misma posición.

Tabla de contenidos

Organización general de la caché



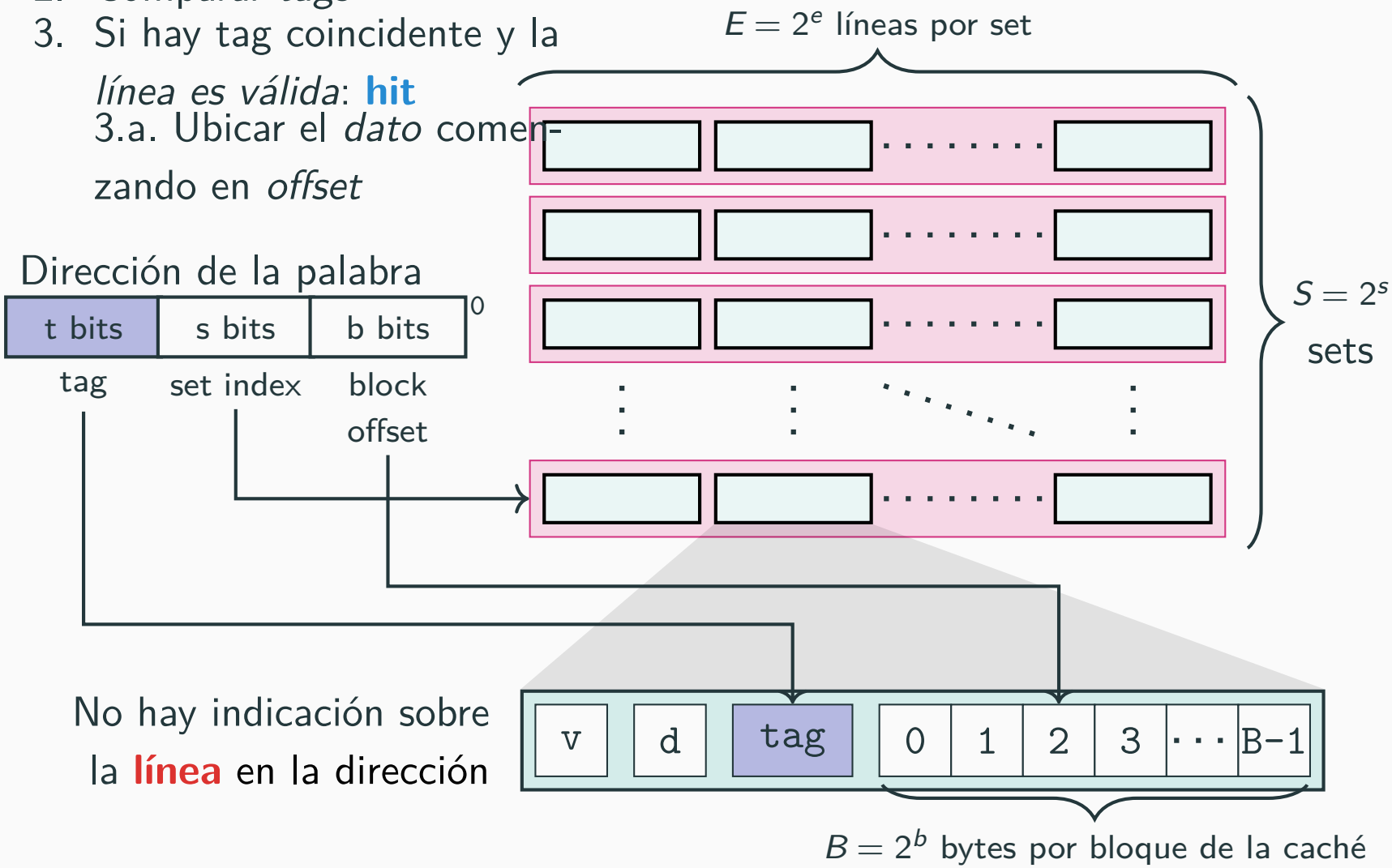
Tamaño de la caché

$S \cdot E \cdot B$ **bytes (datos)**

Tabla de contenidos

Lectura

1. Ubicar el *set*
2. Comparar *tags*
3. Si hay tag coincidente y la *línea es válida*: **hit**
   3.a. Ubicar el *dato* comenzando en *offset*

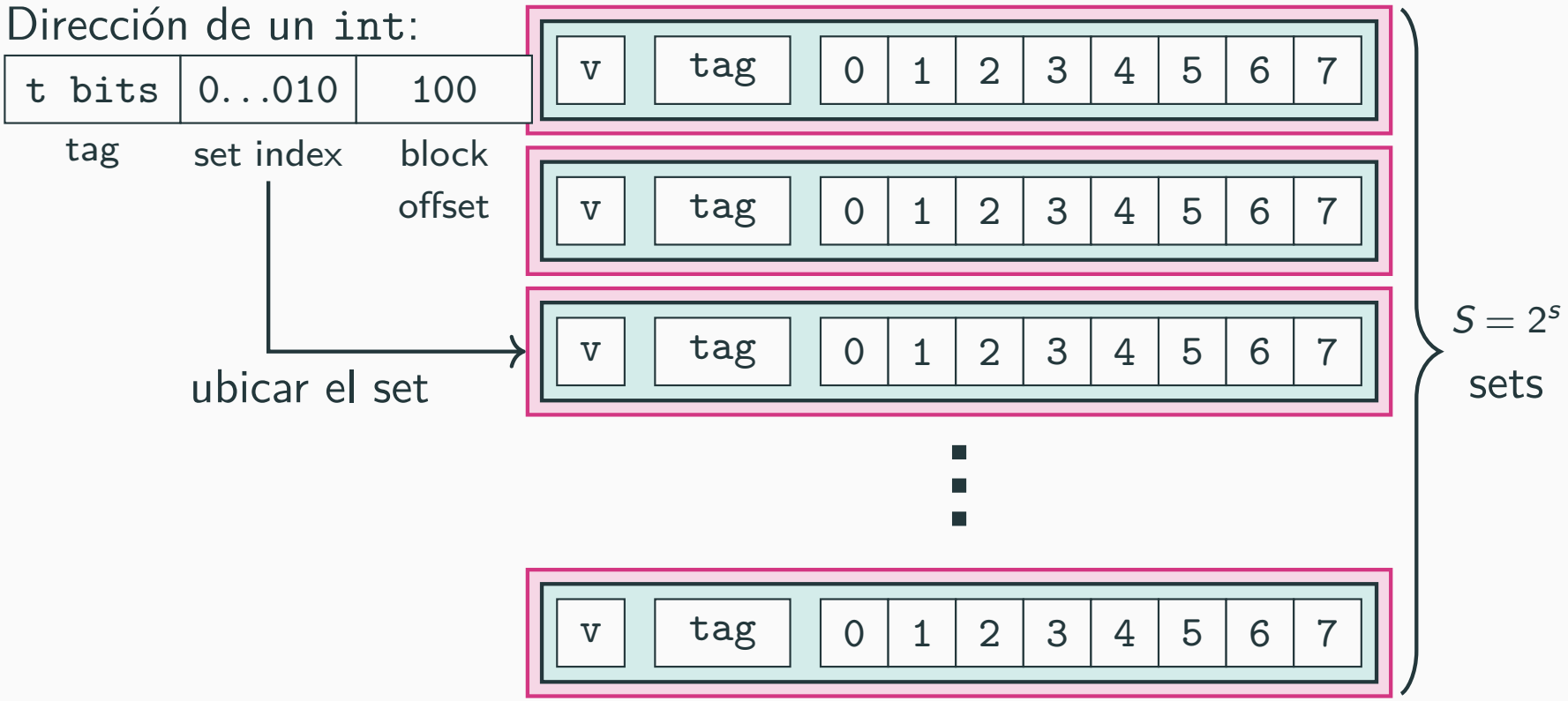


Ejemplo: caché de mapeo directo

Mapeo directo: una línea por cada set

Asumimos un tamaño de bloque de 8 bytes

1. Ubicar el set donde estaría el dato



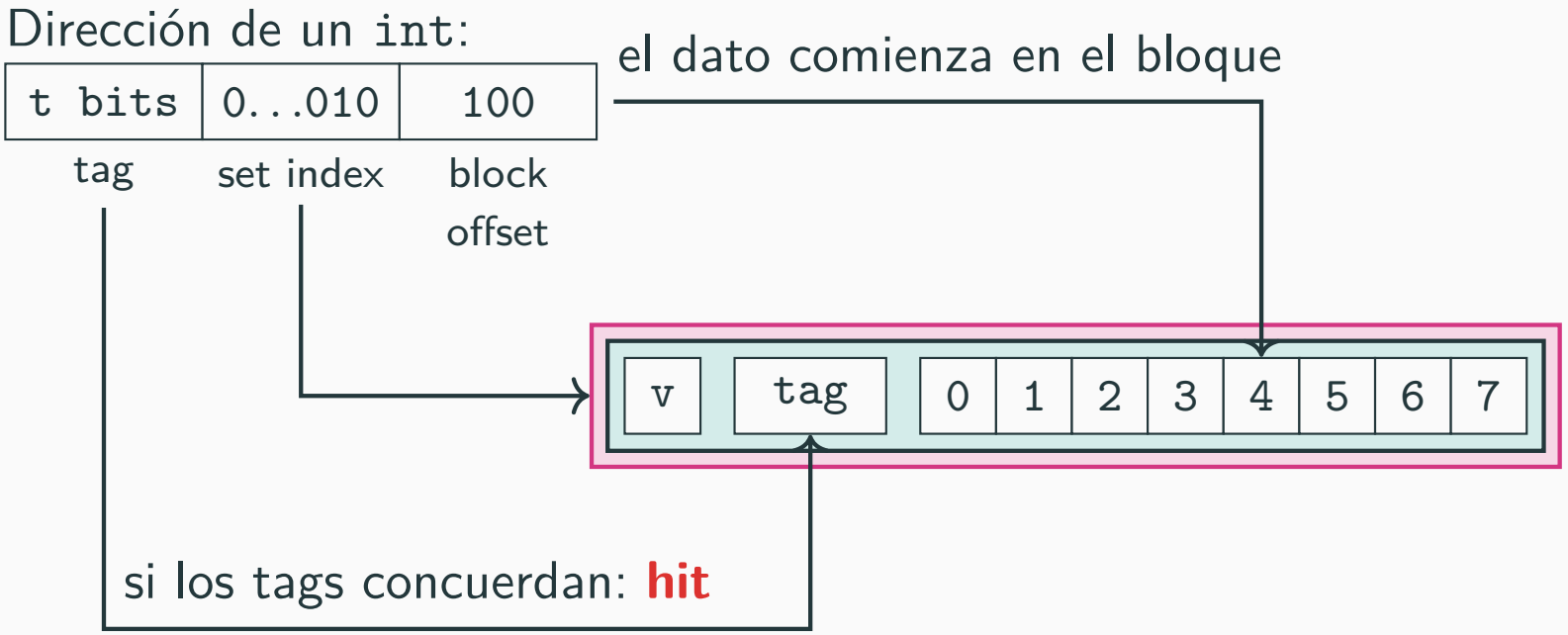
Ejemplo: caché de mapeo directo

Mapeo directo: una línea por cada set

Asumimos un tamaño de bloque de 8 bytes

2a. Chequear el `bit` de **validez**

2b. Comparar los `tags`



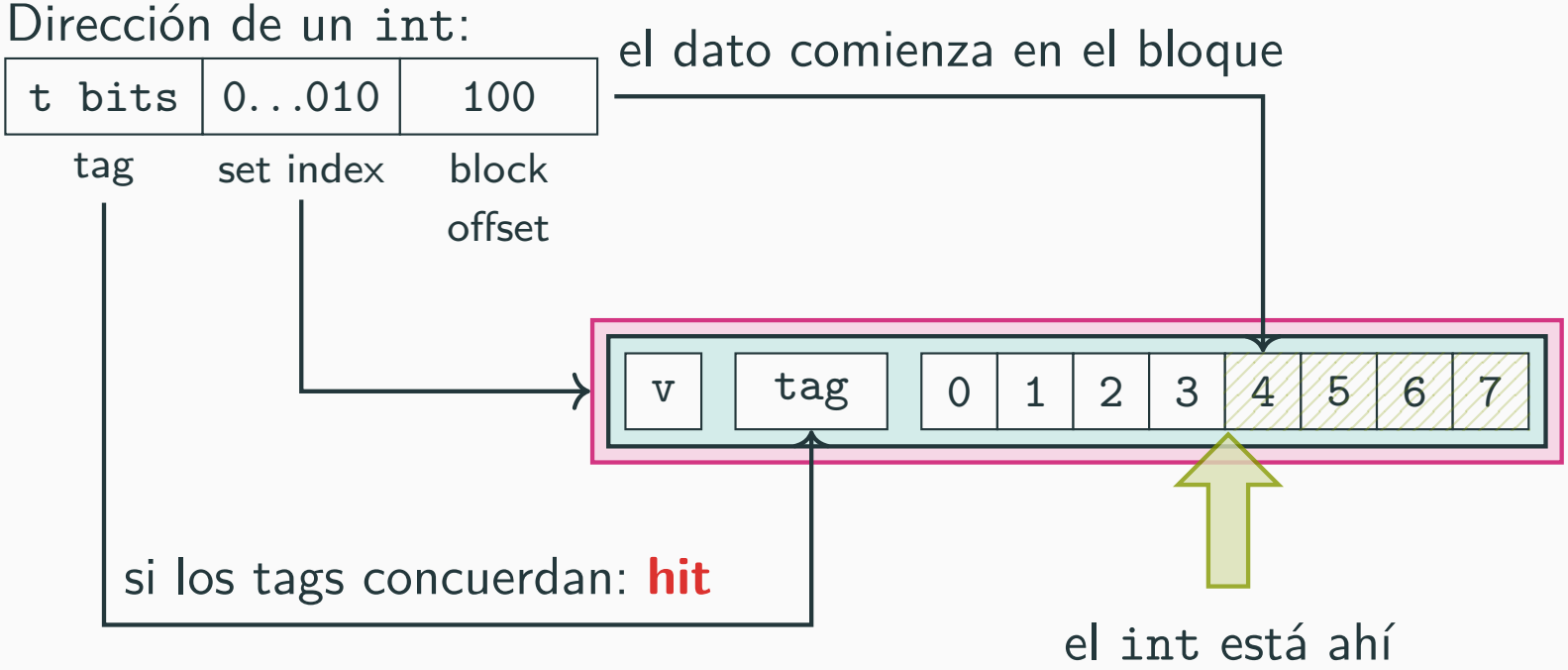
Ejemplo: caché de mapeo directo

Mapeo directo: una línea por cada set

Asumimos un tamaño de bloque de 8 bytes

Si los tags no concuerdan es un **miss**

$\Rightarrow$ la línea es *desalojada* y *reemplazada*



Ejemplo: simulación de lecturas en caché de mapeo directo

Dirección de 4 bits (tamaño del espacio de direcciones: 16 bytes)

Características de la memoria:

- 4 sets (S=4) ⇒ 2 bits, s, para el *set index*,
- 1 línea por set (E=1),
- 2 bytes por bloque (B=2) ⇒ 1 bit, b, para el *block offset*,
- el resto de los bits, t, son para el *tag*.

Seguimiento de accesos a memoria (1 byte por lectura)

	v	tag	bloque
set 0	0	?	?
set 1	0	?	?
set 2	0	?	?
set 3	0	?	?

dirección

hexa	t s b	hit/miss
0x0	0000	
0x1	0001	
0x7	0111	
0x8	1000	
0x0	0000	

Ejemplo: simulación de lecturas en caché de mapeo directo

Dirección de 4 bits (tamaño del espacio de direcciones: 16 bytes)

Características de la memoria:

- 4 sets (S=4) ⇒ 2 bits, `s`, para el *set index*,
- 1 línea por set (E=1),
- 2 bytes por bloque (B=2) ⇒ 1 bit, `b`, para el *block offset*,
- el resto de los bits, `t`, son para el *tag*.

Seguimiento de accesos a memoria (1 byte por lectura)

	v	tag	bloque
set 0	1	0	M[0-1]
set 1	0	?	?
set 2	0	?	?
set 3	1	0	M[6-7]

Esquema final de la memoria

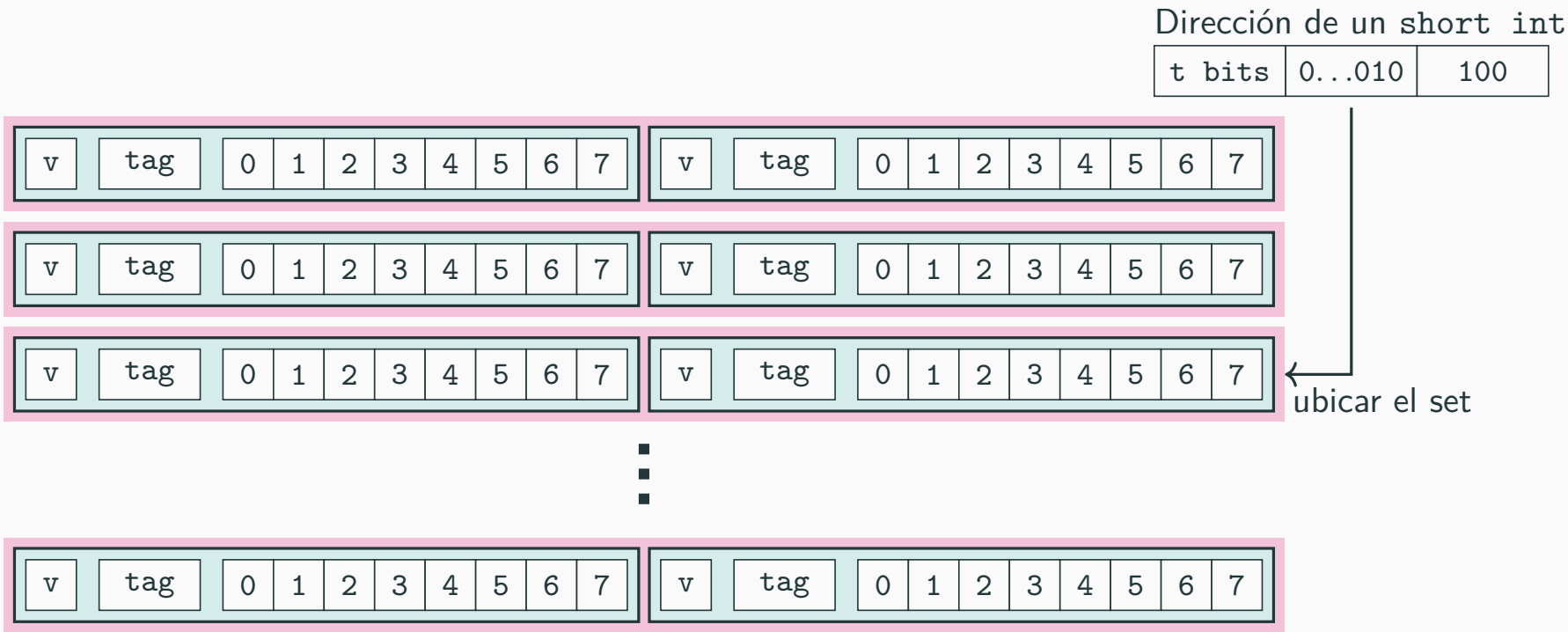
dirección		
hexa	t s b	hit/miss
0x0	0000	miss
0x1	0001	hit
0x7	0111	miss
0x8	1000	miss
0x0	0000	miss

Ejemplo: caché asociativa de E vías

Mapeo asociativo: E líneas por cada set (ejemplo con E=2)

Asumimos un tamaño de bloque de 8 bytes

1. Ubicar el set donde estaría el dato



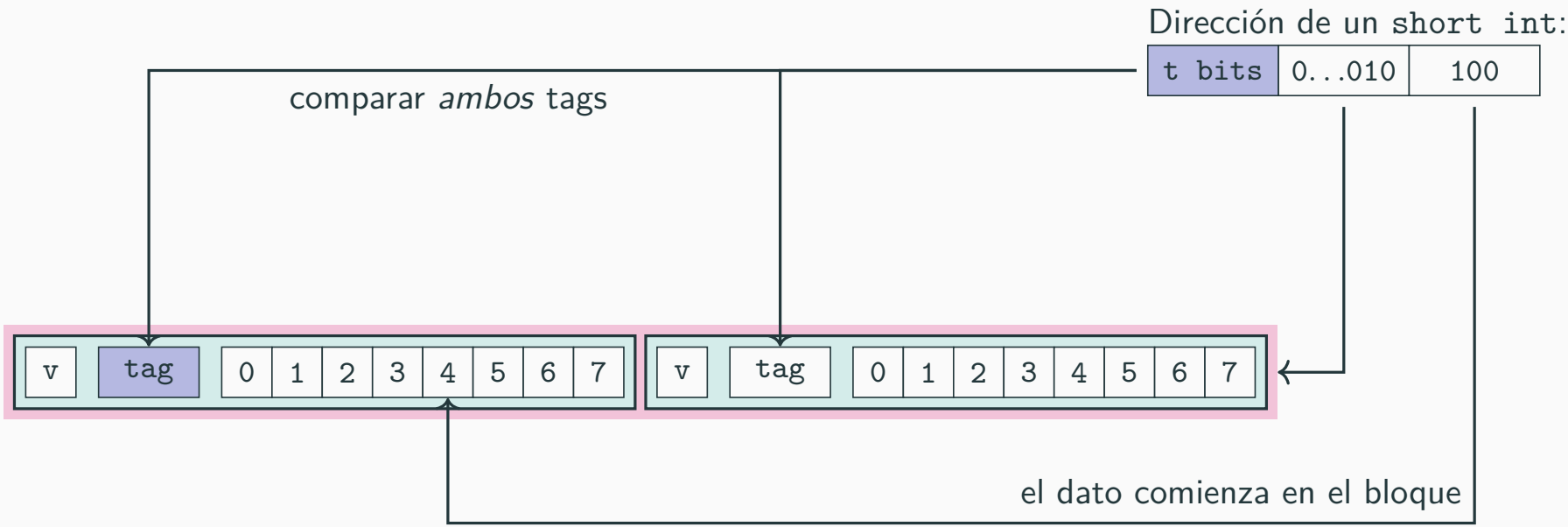
Ejemplo: caché asociativa de E vías

Mapeo asociativo: E líneas por cada set (ejemplo con E=2)

Asumimos un tamaño de bloque de 8 bytes

2a. Chequear el `bit` de **validez**

2b. Comparar los `tags`



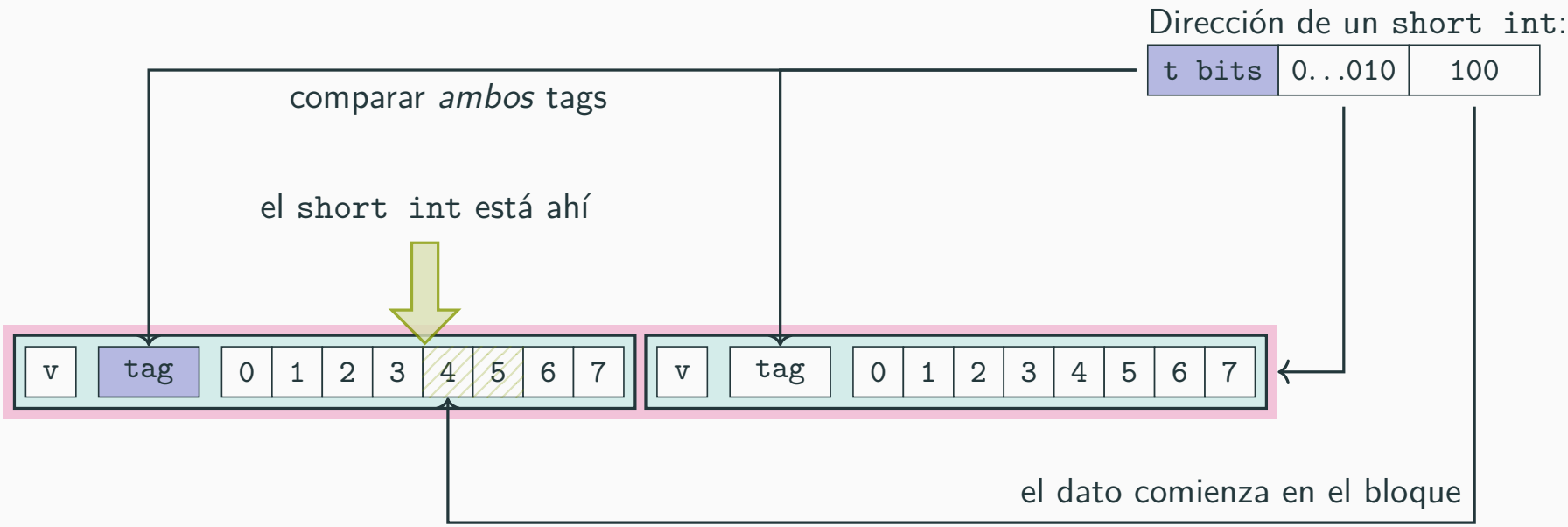
Ejemplo: caché asociativa de E vías

Mapeo asociativo: E líneas por cada set (ejemplo con E=2)

Asumimos un tamaño de bloque de 8 bytes

Si ningún tag concuerda es un **miss**

⇒ una línea es *desalojada* y *reemplazada* **¿cuál?**



Ejemplo: simulación de lecturas en caché asociativa

Dirección de 4 bits (tamaño del espacio de direcciones: 16 bytes)

Características de la memoria:

- 2 sets (S=2) $\Rightarrow$ 1 bit, `s`, para el *set index*,
- 2 líneas por set (E=2),
- 2 bytes por bloque (B=2) $\Rightarrow$ 1 bit, `b`, para el *block offset*,
- el resto de los bits, `t`, son para el *tag*.

Seguimiento de accesos a memoria (1 byte por lectura)

	v	tag	bloque
set 0	0	?	?
	0	?	?
set 1	0	?	?
	0	?	?

dirección		
hexa	t sb	hit/miss
0x0	0000	
0x1	0001	
0x7	0111	
0x8	1000	
0x0	0000	

Ejemplo: simulación de lecturas en caché asociativa

Dirección de 4 bits (tamaño del espacio de direcciones: 16 bytes)

Características de la memoria:

- 2 sets (S=2) ⇒ 1 bit, `s`, para el *set index*,
- 2 líneas por set (E=2),
- 2 bytes por bloque (B=2) ⇒ 1 bit, `b`, para el *block offset*,
- el resto de los bits, `t`, son para el *tag*.

Seguimiento de accesos a memoria (1 byte por lectura)

Esquema final de la memoria

	v	tag	bloque
set 0	1	00	M[0-1]
	1	10	M[8-9]
set 1	1	01	M[6-7]
	0	?	?

dirección		
hexa	t sb	hit/miss
0x0	0000	miss
0x1	0001	hit
0x7	0111	miss
0x8	1000	miss
0x0	0000	hit

Tabla de contenidos

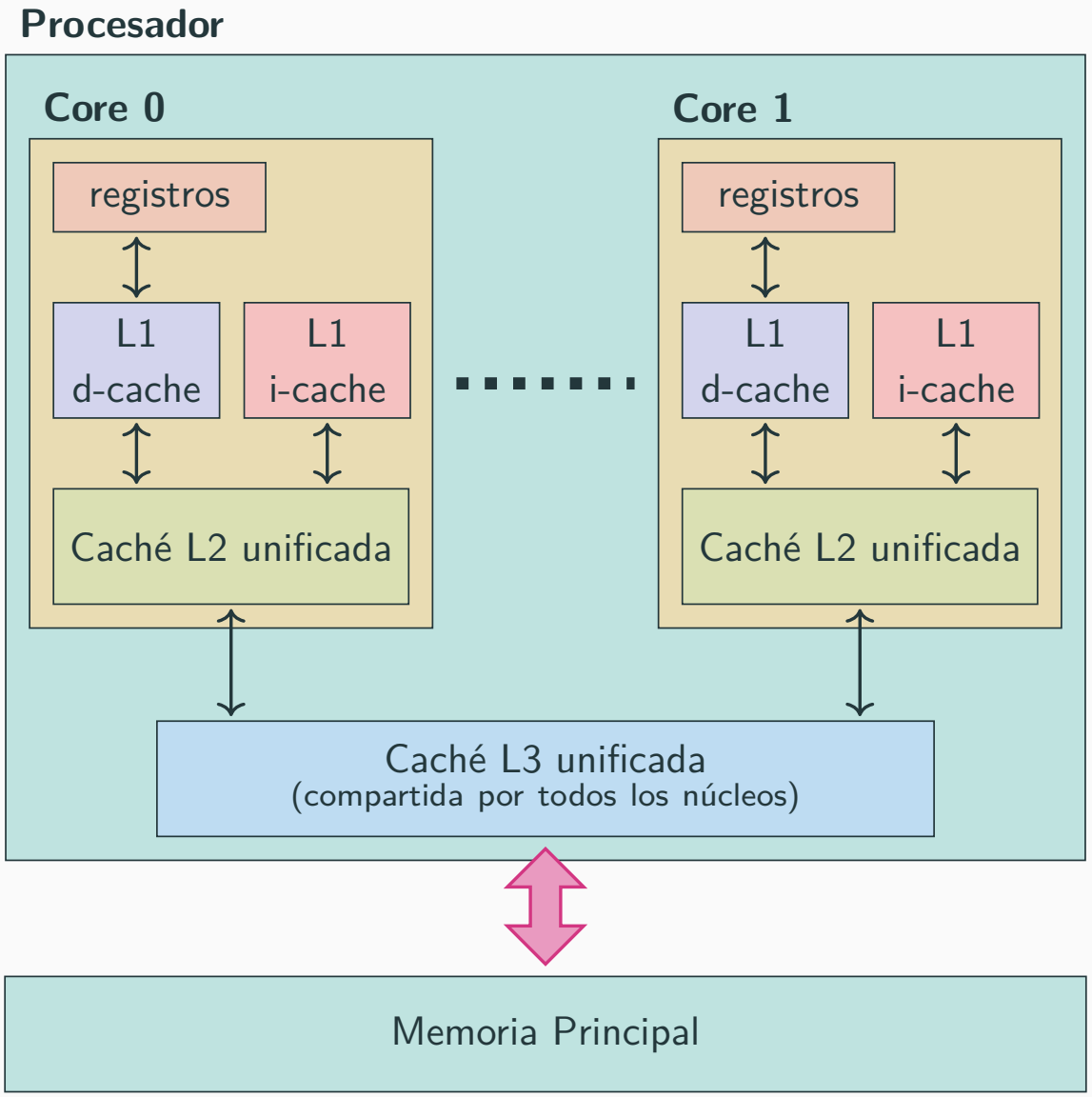
Escritura en la caché

- Existe múltiples copias de los datos $\longrightarrow$ requiere coherencia
  - L1, L2, L3, Memoria Principal, Disco
- ¿Qué se hace — ¿cuál es la política? — cuando hay un *hit* de escritura?
  - **Escritura inmediata** / ***write-through***: escribe el dato en memoria en el momento
  - **Posescritura** / ***write-back***: escribe el dato en memoria cuando se desaloja el bloque (requiere un bit extra)
- ¿Qué se hace cuando hay un *miss* de escritura?
  - ***write-allocate***: se carga la línea en la caché y se escribe
  - ***no-write-allocate***: escribe directamente en memoria
- Cualquier combinación de políticas funciona, pero típicamente se utilizan:
  - ***write-through*** / ***no-write-allocate***
  - ***write-back*** / ***write-allocate***

Ejemplo de escritura *write-back / write-allocate*

- Se emite una escritura en el la dirección X
- Si es un *hit*/acierto:
  - Se actualiza el contenido del bloque
  - Se pone el *dirty* bit en 1
- Si es un *miss*/fallo:
  - Se trae el bloque de memoria (como en un fallo de lectura)
  - Se emite una escritura (que es un acierto)
- Si una línea es desalojada y posee el *dirty* bit en 1:
  - Se escribe en memoria el bloque completo (2^b bytes)
  - Se limpia el *dirty* bit (se pone en 0)
  - Se reemplaza la línea con el nuevo contenido

Organización de la memoria de un Intel Core i7



- **Cachés L1:**
  - 32 KB, 8 vías,
  - Acceso: 4 ciclos
- **Caché L2:**
  - 256 KB, 8 vías,
  - Acceso: 10 ciclos
- **Caché L3:**
  - 8 MB, 16 vías,
  - Acceso: 40-75 ciclos
- **Tamaño del bloque:**
  - 64 bytes para todas las cachés

Tabla de contenidos

Métricas de desempeño

- **Tasa de aciertos / *Hit rate***
  - fracción de las referencias a memoria que están en la caché (aciertos / pedidos, *hits* / *accesses*).
- **Tasa de fallos / *Miss rate***
  - fracción de las referencias a memoria que **no** están en la caché (fallos / pedidos, *misses* / *accesses*) $= 1 -$ hit rate.
  - típicamente: 3 % a 10 % para L1, menor para L2 (incluso < 1 %).
- **Tiempo de acceso / *Hit time***
  - Tiempo que tarda el procesador en obtener una línea de caché.
  - Valores típicos: 4 ciclos para L1, 10 ciclos para L2.
- **Penalización por fallo / *Miss penalty***
  - Tiempo adicional requerido debido a un *miss*.
  - Típicamente: 50 c a 200 ciclos para la memoria principal.

Métricas de desempeño

- **La relación entre los tiempos de acceso ante un hit y un miss es muy grande**
  - Puede llegar a 100 veces, considerando únicamente L1 y memoria principal,
  - Todo debido al *miss penalty*.
- 99 % de **hits** es el doble de mejor que 97 % de **hits**
  - Supongamos un *hit time* de 1 ciclo, y un *miss penalty* de 100 ciclos.
  - Tiempos de acceso promedio:
    97 % de aciertos: $1 \text{ ciclo} + 0{,}03 \cdot 100 \text{ ciclos} = 4 \text{ ciclos}$
    99 % de aciertos: $1 \text{ ciclo} + 0{,}01 \cdot 100 \text{ ciclos} = 2 \text{ ciclos}$
- **por eso se usa el *miss rate***

¿Cómo escribir código que sea amigable con la caché?

- Hacer **rápido el caso común**
  - Hacer foco en los ciclos de las funciones principales, de adentro hacia afuera.

- Minimizar los **misses** en los ciclos internos
  - Referenciar variables repetidamente (localidad temporal)
  - El patrón de acceso de 1 paso (*stride-1*) es bueno (localidad espacial)

Ejemplo: caché en frío, palabras de 4 bytes, bloques de 4 palabras:

```
1 int sumarrayrows(int a[M][N]) {
2     int i, j, sum = 0;
3     for (i = 0; i < M; i++)
4         for (j = 0; j < N; j++)
5             sum += a[i][j];
6     return sum; }
```

Miss rate: 25 %

```
1 int sumarrayrows(int a[M][N]) {
2     int i, j, sum = 0;
3     for (j = 0; j < N; j++)
4         for (i = 0; i < M; i++)
5             sum += a[i][j];
6     return sum; }
```

Miss rate: 100 %

Tabla de contenidos

The Memory Mountain

- *Throughput* de lectura (ancho de banda de lectura)
  - Cantidad de bytes leídos de la memoria por segundo (MB/s)
- *Memory mountain*: *throughput* de lectura leído en función de la localidad espacial y la localidad temporal.
  - Es una forma compacta de caracterizar, mediante un gráfico, el desempeño de un sistema en cuanto a la memoria.

Función de pruebas para la montaña de memoria

```
1  long data[MAXELEMS]; /* The global array we'll be traversing */
2  /* test - Iterate over first "elems" elements of array "data"
3   *        with stride of "stride", using 4x4 loop unrolling. */
4  int test(int elems, int stride) {
5      long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
6      long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
7      long length = elems;
8      long limit = length - sx4;
9
10     /* Combine 4 elements at a time */
11     for (i = 0; i < limit; i += sx4) {
12         acc0 = acc0 + data[i];
13         acc1 = acc1 + data[i+stride];
14         acc2 = acc2 + data[i+sx2];
15         acc3 = acc3 + data[i+sx3];
16     }
17
18     /* Finish any remaining elements */
19     for (; i < length; i += stride) {
20     acc0 = acc0 + data[i];
21     }
22     return ((acc0 + acc1) + (acc2 + acc3));
23 }
```

La Montaña

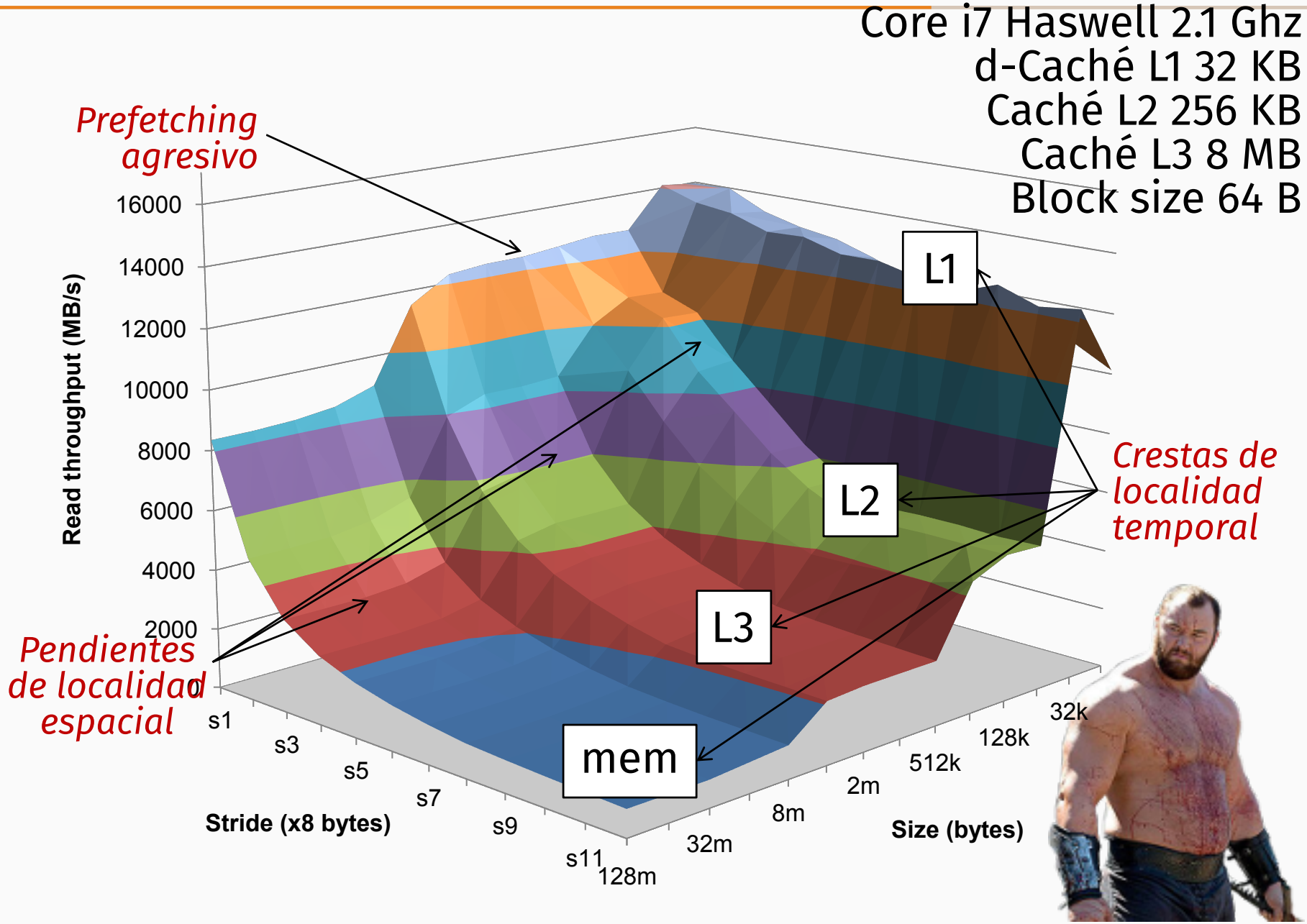

Core i7 Haswell 2.1 Ghz
d-Caché L1 32 KB
Caché L2 256 KB
Caché L3 8 MB
Block size 64 B
Prefetching agresivo
Read throughput (MB/s)
16000
14000
12000
10000
8000
6000
4000
2000
0
L1
L2
L3
mem
Crestas de localidad temporal
Pendientes de localidad espacial
s1
s3
s5
s7
s9
s11
Stride (x8 bytes)
32k
128k
512k
2m
8m
32m
128m
Size (bytes)

Tabla de contenidos

Ejemplo: multiplicación de matrices

Elementos a considerar

- Tamaño de la caché
- Tamaño del bloque
- Orden de los 3 bucles

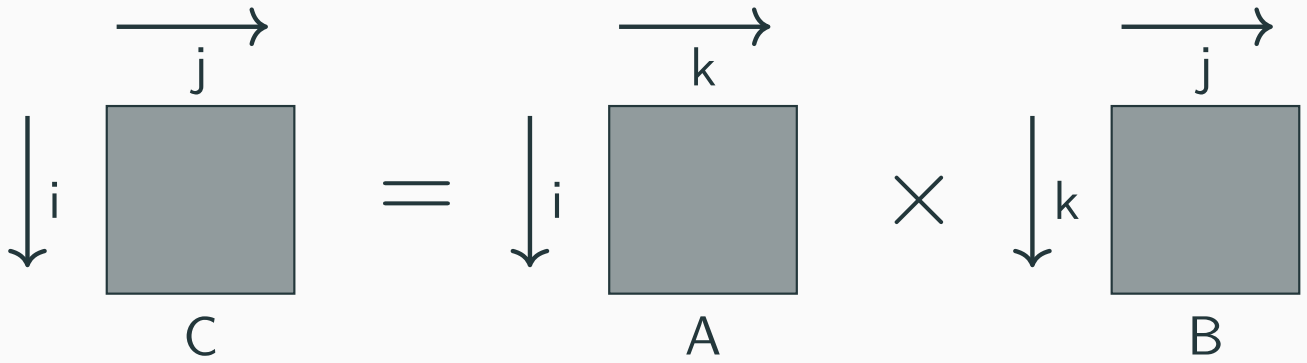
```
/* ijk */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Descripción

- Multiplicar matrices $N \times N$
- Matrices de doubles (8 bytes)
- Operaciones: $O(N^3)$
- N lecturas por elemento de origen
- N valores sumados por destino

Análisis de la tasa de fallos (*miss rate*)

- **Asumimos:**
  - Tamaño de bloque: 32 bytes (alcanza para 4 `doubles`)
  - Dimensión de la matriz muy grande: $1/N \to 0{,}0$
  - La caché no tiene tamaño suficiente para guardar múltiples filas
- **Método para el análisis**
  - Examinar los bucles internos

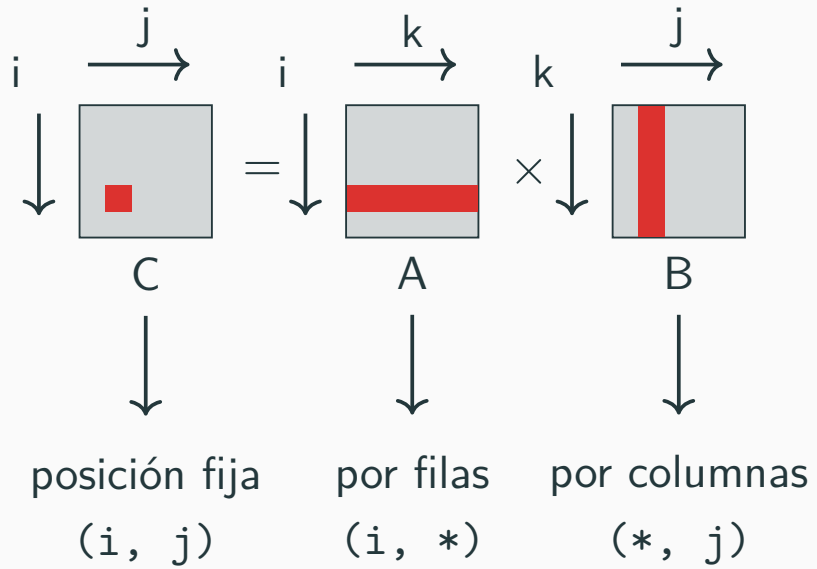


Disposición de los arreglos de C en memoria

- **Los arreglos en C se almacenan en *row-major order***
  - elementos contiguos de una fila están en ubicaciones contiguas
  - las filas, como bloques de memoria, están en posiciones contiguas
- **Avanzando por las columnas de una fila**
  - ```
    for (i = 0; i < N; i++)
        sum += a[0][i];
    ```
  - accede a elementos consecutivos
  - si el tamaño del bloque (B) $>$ `sizeof(`a_{ij}`)`, aprovecha la localidad espacial: **miss rate** $=$ `sizeof(`a_{ij}`)`$/$B
- **Avanzando por las filas de una columna**
  - ```
    for (i = 0; i < N; i++)
        sum += a[i][0];
    ```
  - accede a elementos distantes en memoria (N grande)
  - no hay localidad espacial: **miss rate** $= 1$ (100 %)

Multiplicación de matrices (`ijk`)

```
/* ijk */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```



Tasa de fallos en el bucle interno:

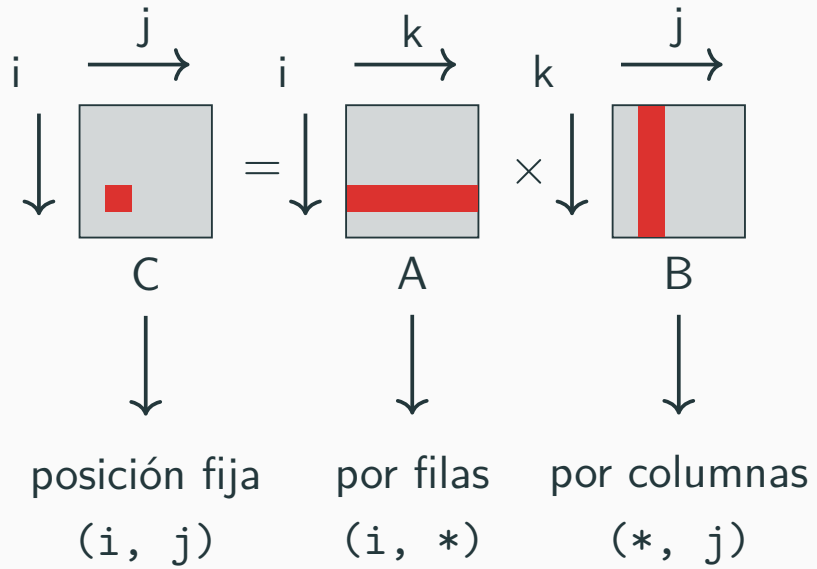
C A B

`ijk (jik)`:

- 2 cargas, 0 almacenamientos
- promedio de fallas por iteración:

Multiplicación de matrices (`ijk`)

```
/* ijk */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```



Tasa de fallos en el bucle interno:

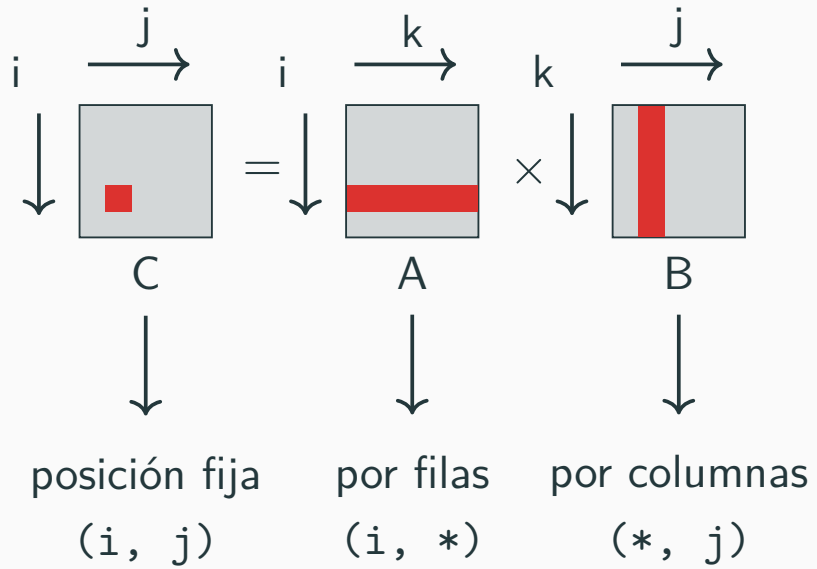
C	A	B
0.00	0.25	1.00

`ijk (jik)`:

- 2 cargas, 0 almacenamientos
- promedio de fallas por iteración:

Multiplicación de matrices (`ijk`)

```
/* ijk */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```



Tasa de fallos en el bucle interno:

C	A	B
0.00	0.25	1.00

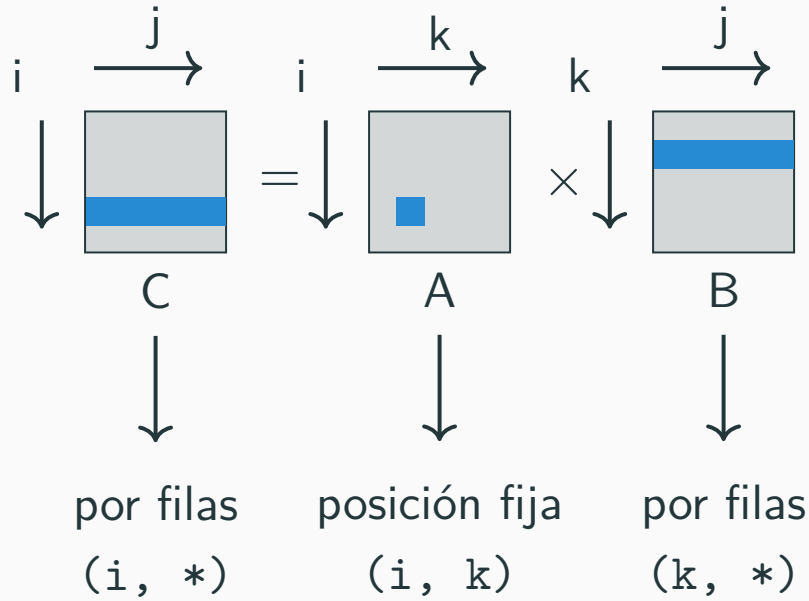
`ijk (jik)`:

- 2 cargas, 0 almacenamientos
- promedio de fallas por iteración: **1.25**

Multiplicación de matrices (`kij`)

```
/* kij */
for (k = 0; k < n; k++) {
    for (i = 0; i < n; i++) {
        r = a[i][k]
        for (j = 0; j < n; j++)
            c[i][j] += r * b[k][j];

    }
}
```



Tasa de fallos en el bucle interno:

C A B

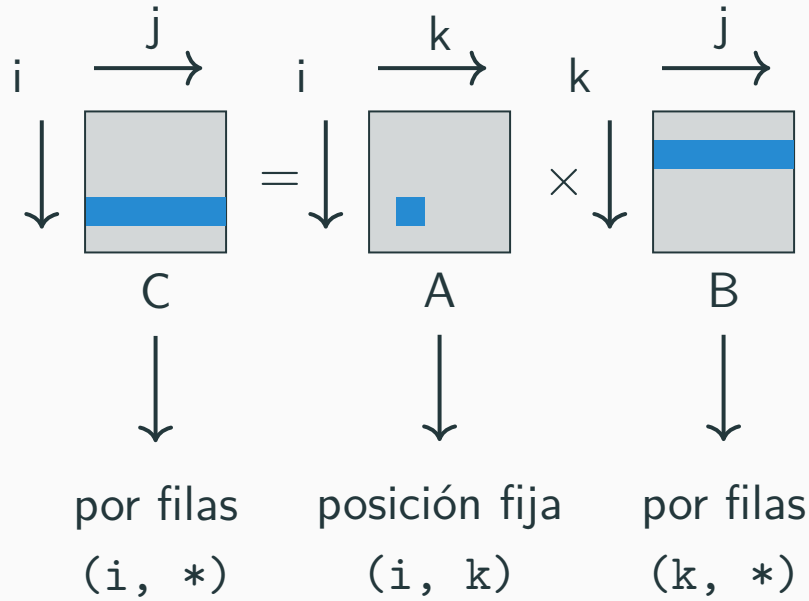
`kij (ikj):`

- 2 cargas, 1 almacenamiento
- promedio de fallas por iteración:

Multiplicación de matrices (kij)

```
/* kij */
for (k = 0; k < n; k++) {
    for (i = 0; i < n; i++) {
        r = a[i][k]
        for (j = 0; j < n; j++)
            c[i][j] += r * b[k][j];

    }
}
```



Tasa de fallos en el bucle interno:

C	A	B
0.25	0.00	0.25

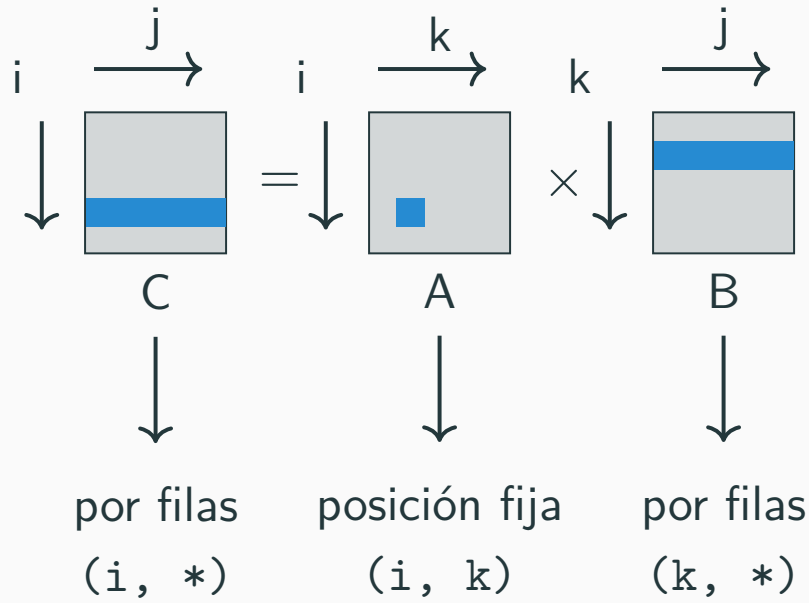
kij (ikj):

- 2 cargas, 1 almacenamiento
- promedio de fallas por iteración:

Multiplicación de matrices (`kij`)

```
/* kij */
for (k = 0; k < n; k++) {
    for (i = 0; i < n; i++) {
        r = a[i][k]
        for (j = 0; j < n; j++)
            c[i][j] += r * b[k][j];

    }
}
```



Tasa de fallos en el bucle interno:

C	A	B
0.25	0.00	0.25

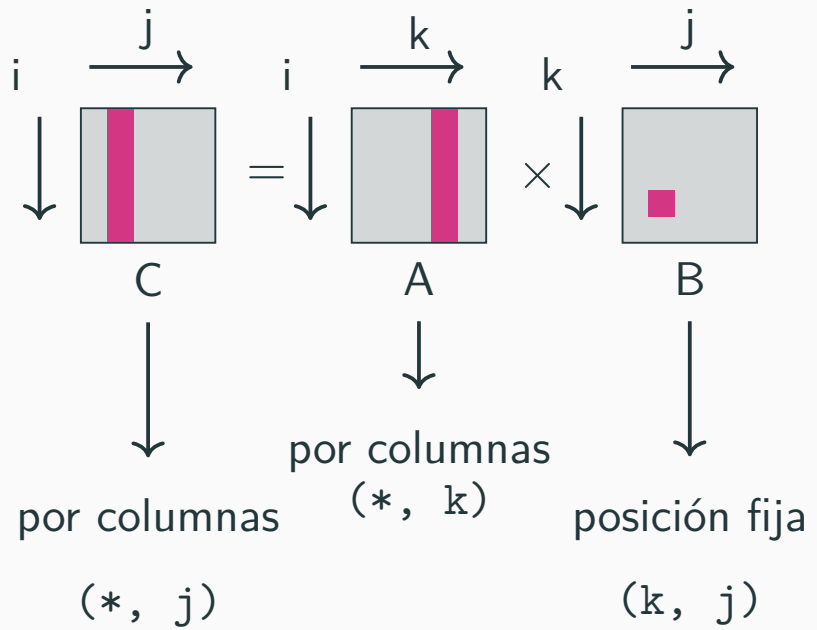
`kij (ikj)`:

- 2 cargas, 1 almacenamiento
- promedio de fallas por iteración: **0.50**

Multiplicación de matrices (`jki`)

```
/* jki */
for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
        r = b[k][j]
        for (i = 0; i < n; i++)
            c[i][j] += a[i][k] * r;

    }
}
```



Tasa de fallos en el bucle interno:

C A B

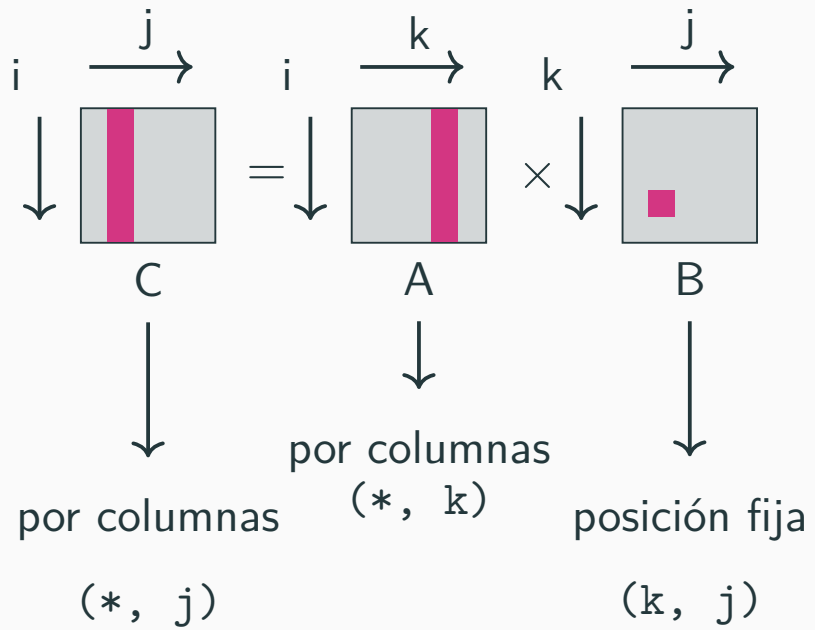
`jki (kji)`:

- 2 cargas, 1 almacenamiento
- promedio de fallas por iteración:

Multiplicación de matrices (`jki`)

```
/* jki */
for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
        r = b[k][j]
        for (i = 0; i < n; i++)
            c[i][j] += a[i][k] * r;

    }
}
```



Tasa de fallos en el bucle interno:

C	A	B
1.00	0.00	1.00

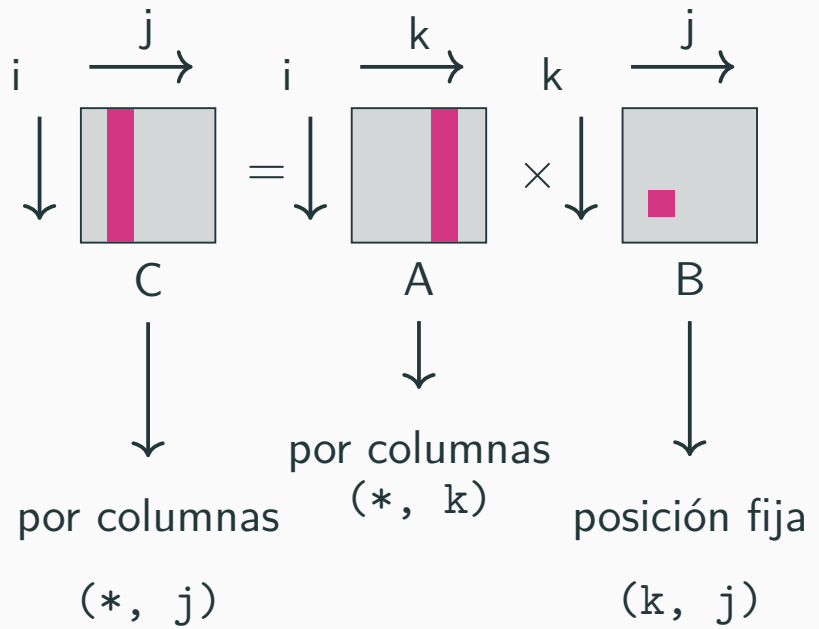
`jki (kji)`:

- 2 cargas, 1 almacenamiento
- promedio de fallas por iteración:

Multiplicación de matrices (`jki`)

```
/* jki */
for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
        r = b[k][j]
        for (i = 0; i < n; i++)
            c[i][j] += a[i][k] * r;

    }
}
```



Tasa de fallos en el bucle interno:

C	A	B
1.00	0.00	1.00

`jki (kji)`:

- 2 cargas, 1 almacenamiento
- promedio de fallas por iteración: **2.00**

Desempeño de la multiplicación matricial en un Core i7

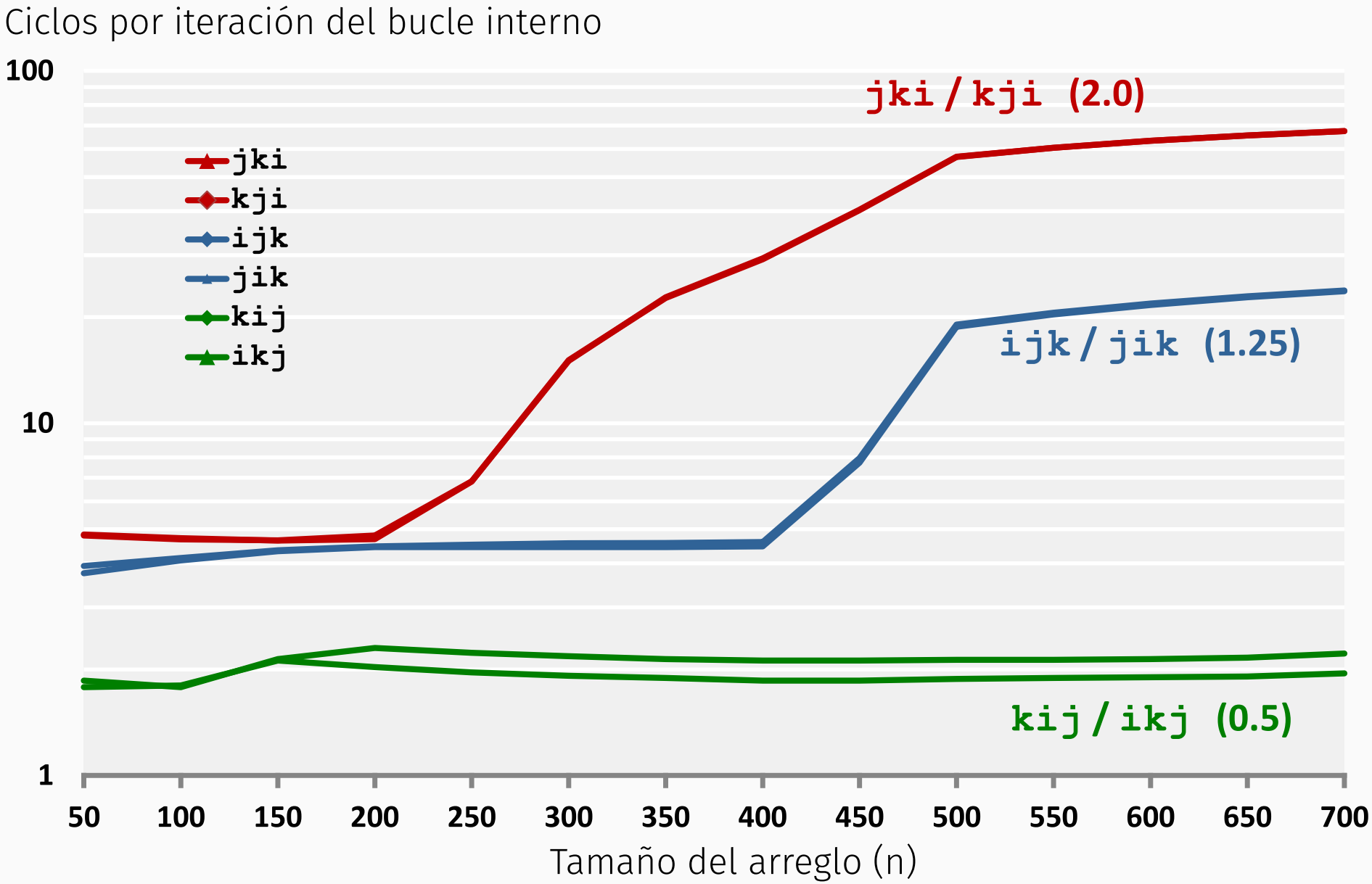


Tabla de contenidos

Blocking

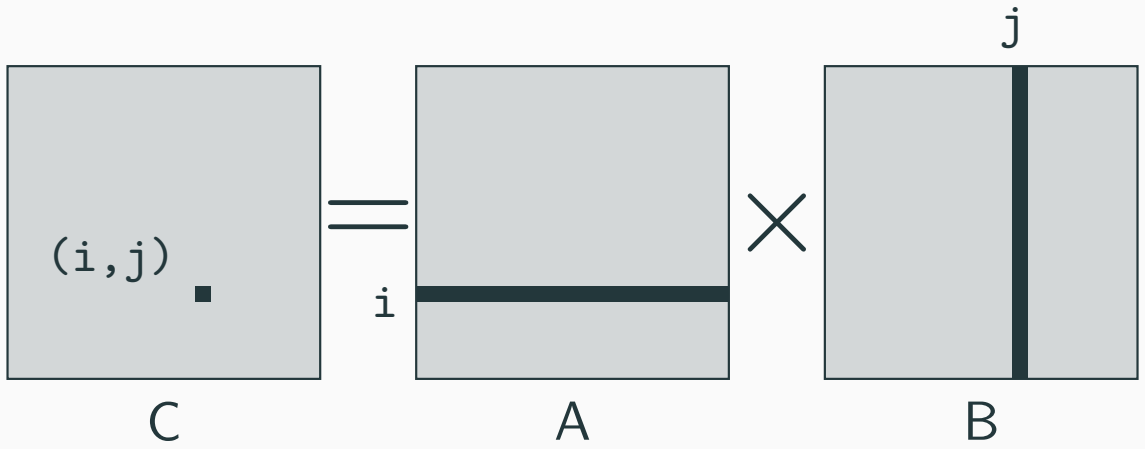
Blocking es una técnica que se utiliza para aprovechar la localidad de la información en los ciclos anidados. Citando al trabajo *The Cache Performance and Optimizations of Blocked Algorithms*[1]:

> *En vez de operar sobre filas o columnas completas de un arreglo, los algoritmos* bloqueados *trabajan con submatrices o* bloques, *de forma tal que los datos cargados en los niveles más rápidos de la jerarquía de memoria son reutilizados.*

[1] Monica D. Lam, Edward E. Rothberg, y Michael E. Wolf. 1991. «The cache performance and optimizations of blocked algorithms». En: *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. Association for Computing Machinery, New York, NY, USA, 63–74. DOI: https://10.1145/106972.106981

Multiplicación matricial

```
c = (double *) calloc (sizeof(double), n*n);
/* multiplica las matrices nxn `a` y `b` */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n +k] + b[k*n + j];
}
```



Análisis de la tasa de fallos (*miss rate*)

Asumimos:

- Los elementos de la matriz son `doubles`
- Tamaño de los bloques en la caché: 8 `doubles`
- Tamaño de la caché $C \ll n$ (mucho menor a n)

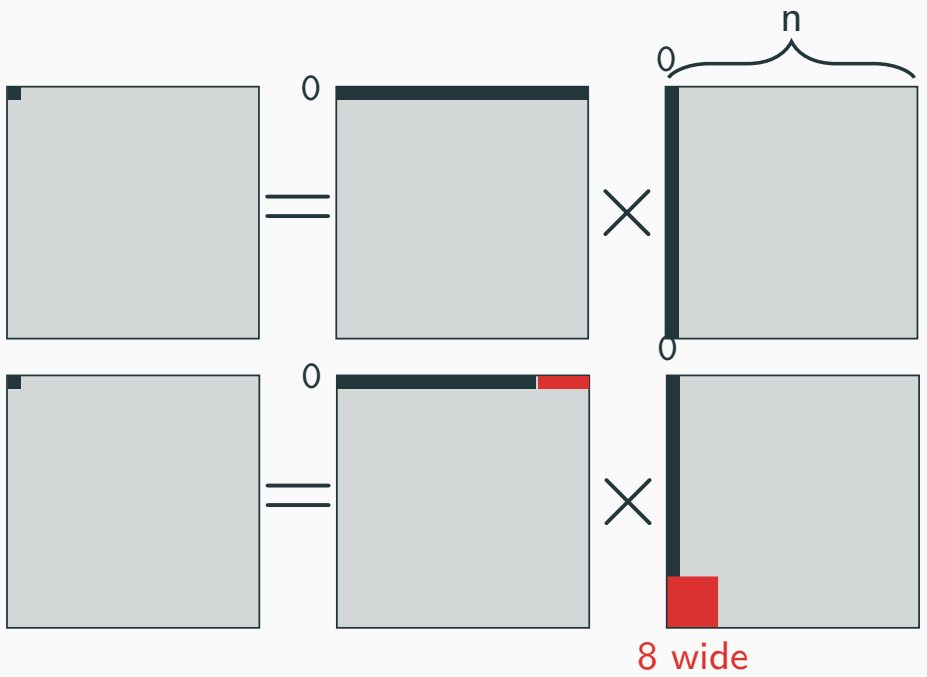
Primera iteración:

- $n/8 + n = 9n/8$ fallas

  1 cada 8 accesos en A

  n en B

- Lo que queda en la caché (en rojo)



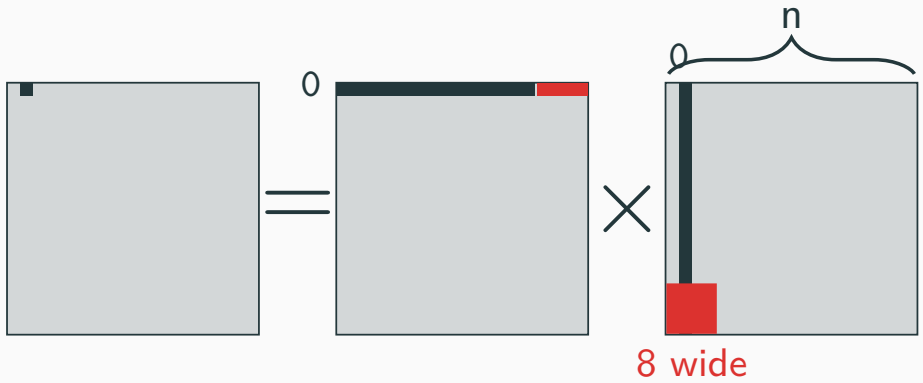
Análisis de la tasa de fallos (*miss rate*)

Asumimos:

- Los elementos de la matriz son `doubles`
- Tamaño de los bloques en la caché: 8 `doubles`
- Tamaño de la caché $C \ll n$ (mucho menor a n)

Segunda iteración:

- Igual a la primera
- $\frac{n}{8} + n = \frac{9}{8}n$ fallas

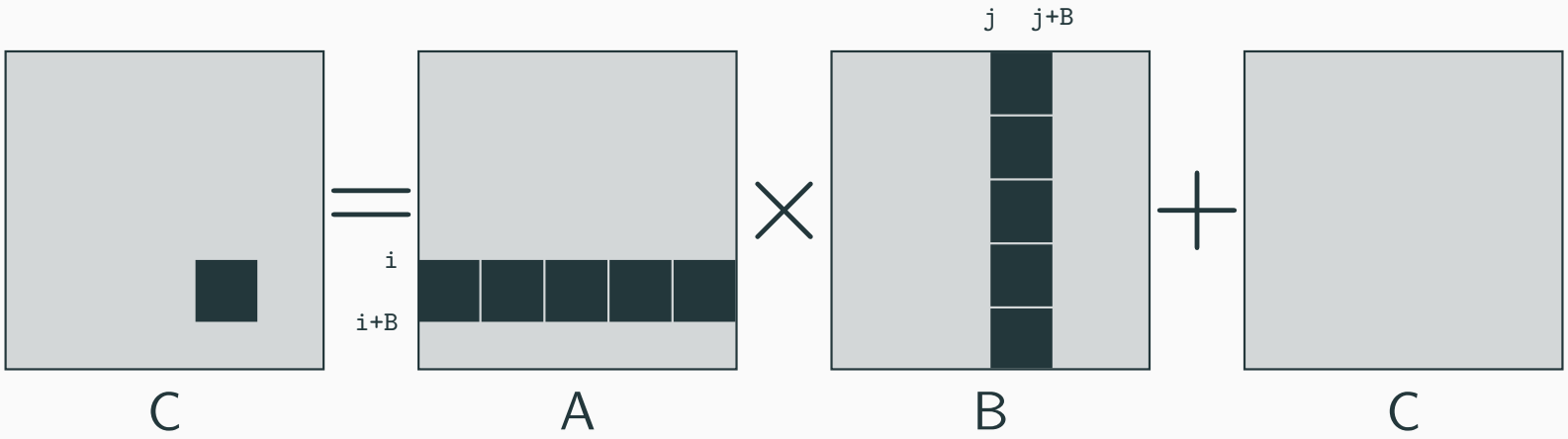


Fallas totales:

- Fallas por cada fila: $9n/8 \cdot n$ (n veces lo anterior)
- Hay n filas, total: $9n/8 \cdot n \cdot n = \frac{9}{8}n^3$

Multiplicación matricial por bloques

```
c = (double *) calloc (sizeof(double), n*n);
/* multiplica las matrices nxn `a` y `b` */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i += B)
        for (j = 0; j < n; j += B)
            for (k = 0; k < n; k += B)
                /* multiplicación de mini matrices BxB */
                for (i1 = 0; i1 < i+B; i1++)
                    for (j1 = 0; j1 < j+B; j1++)
                        for (k1 = 0; k < k+B; k1++)
                            c[i1*n + j1] += a[i1*n + k1] + b[k1*n + j1];
}
```



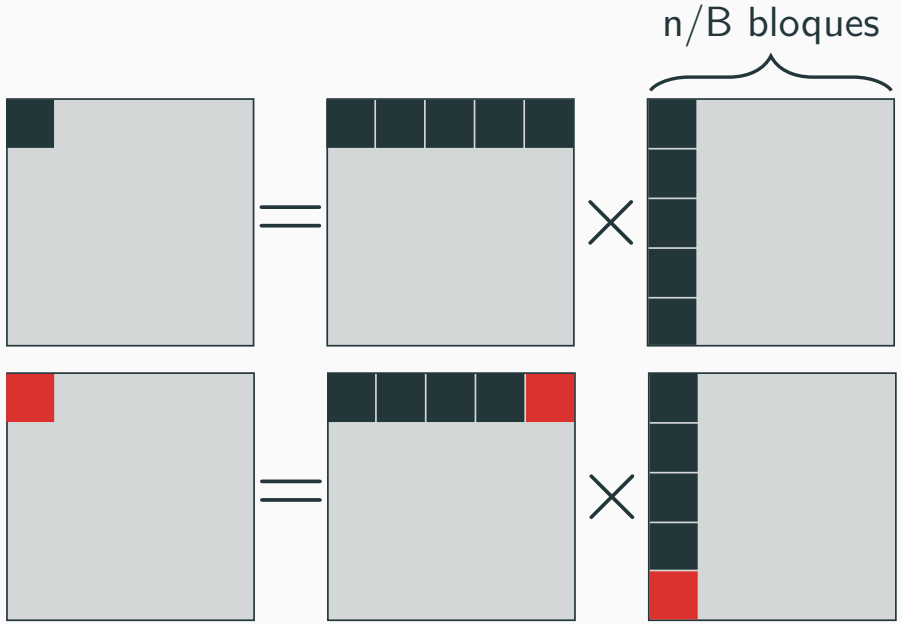
Blocking

Asumimos:

- Tamaño de los bloques en la caché: 8 `doubles`
- Tamaño de la caché $C \ll n$ (mucho menor a n)
- En la caché entran 3 bloques ■ : $3B^2 < C$

Primera iteración en bloque:

- $B^2/8$ fallas por bloque
- $\frac{2n}{B} \cdot \frac{B^2}{8} = \frac{nB}{4}$
- Lo que queda en la caché (en rojo)



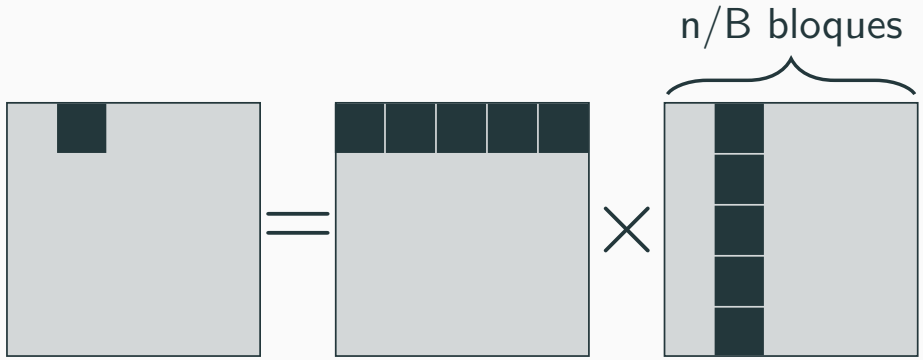
Análisis de la tasa de fallos (*miss rate*)

Asumimos:

- Tamaño de los bloques en la caché: 8 `doubles`
- Tamaño de la caché $C \ll n$ (mucho menor a n)
- En la caché entran 3 bloques ■ : $3B^2 < C$

Segunda iteración:

- Igual a la primera
- $\frac{2n}{B} \cdot \frac{B^2}{8} = \frac{nB}{4}$



Fallas totales:

- $\frac{nB}{4} \cdot \frac{n^2}{B} = \frac{1}{4B} n^3$

Resumen

	Sin *blocking*	Con *blocking*
Fallas (misses)	$\frac{9}{8}n^3$	$\frac{1}{4B}n^3$

- **Elegir el tamaño B más grande que satisfaga $3B^2 < C$**
  - Meter 3 bloques en la caché: 2 entradas + 1 salida
- **¿por qué hay tanta diferencia?**
  - La multiplicación matricial tiene localidad temporal
    - Datos de entrada: $3n^2$, cómputo: $2n^3$
    - Cada elemento de cada matriz se usa $O(n)$ veces
- **En general:**
  - Analizar el algoritmo y usar todos los datos que se cargan en la caché (maximizar localidad temporal)

Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`