Roadmap

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Optimizations
Memory & caches

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

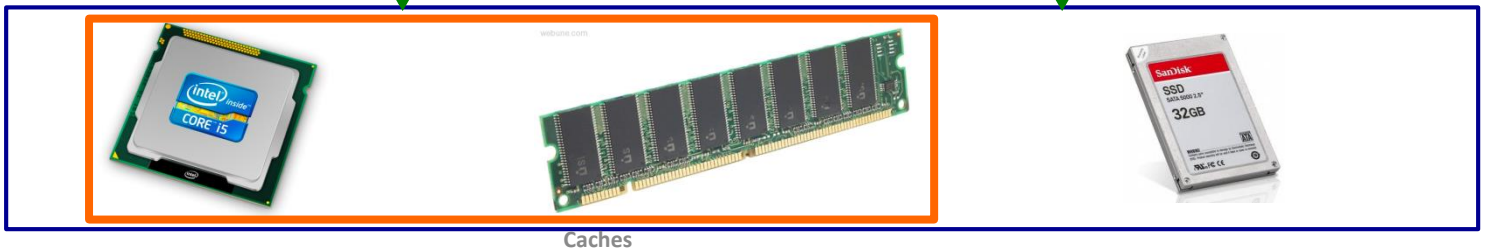
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111101000011111
```

OS:

Computer system:



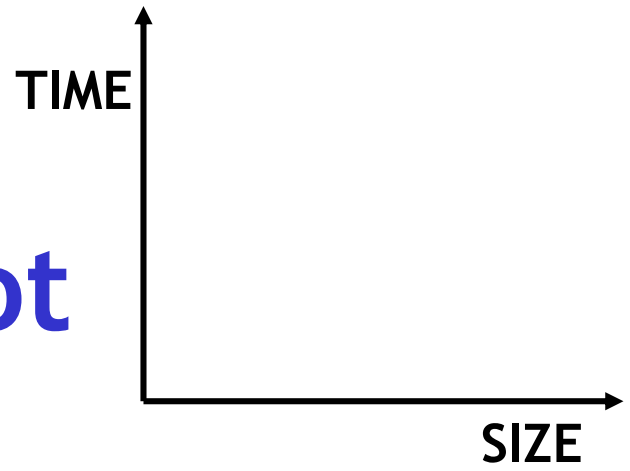
Caches

Cours 8: Mémoire et Caches

- Principe de localité
- Hiérarchie de mémoire
- Organisation de cache
- Optimisations de programme avec cache

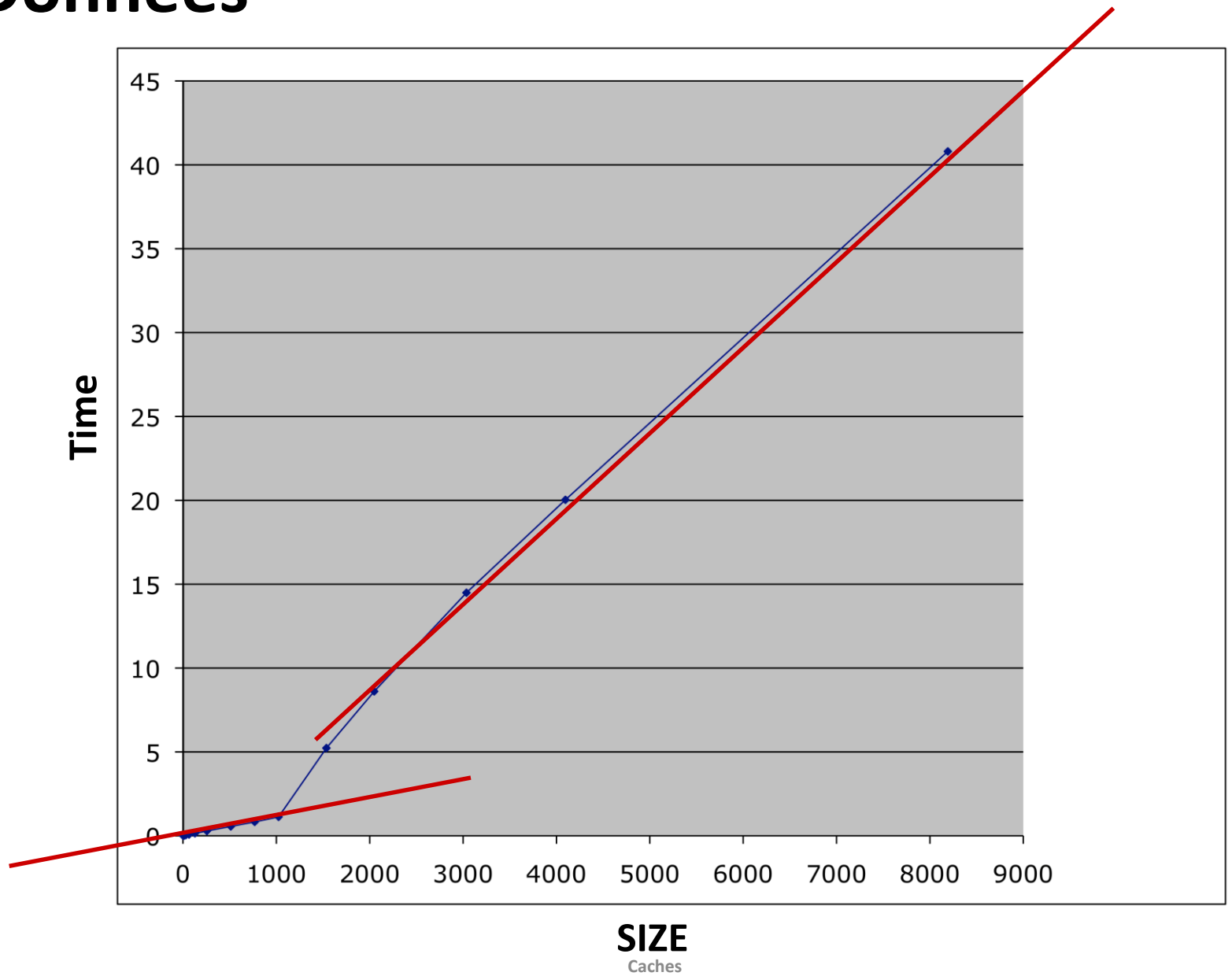
Comment le temps d'exécution augmente avec SIZE?

```
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
  for (int j = 0 ; j < SIZE ; ++ j) {
     A += array[j];
  }
}
```



Plot

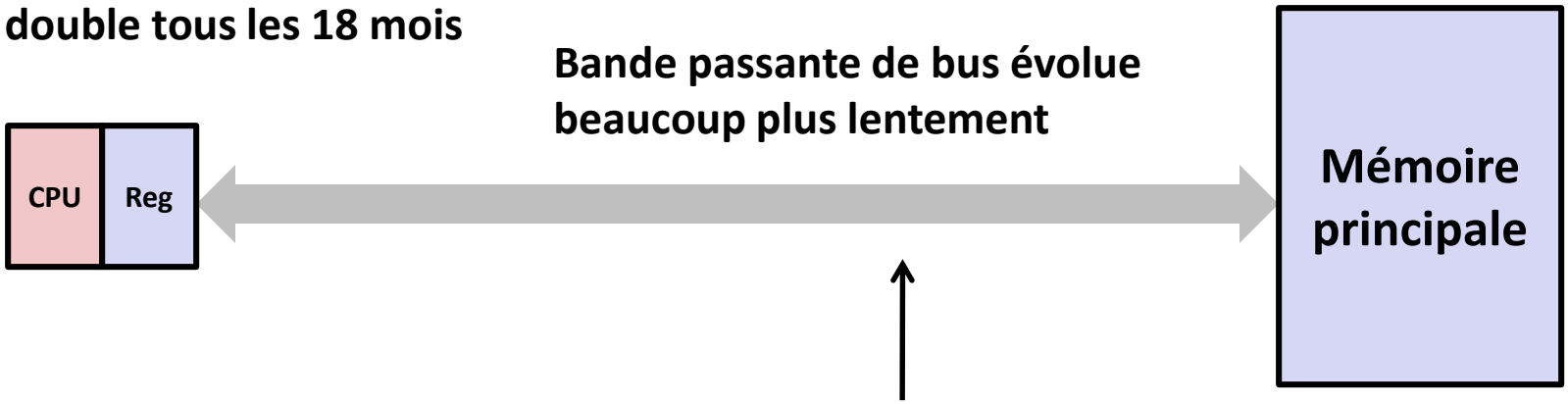
Données


45
40
35
30
25
20
15
10
5
0
Time
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
SIZE

Goulot d'étranglement : processeur - mémoire

La performance de processeur double tous les 18 mois

Bande passante de bus évolue beaucoup plus lentement



Core 2 Duo:
256 Bytes/cycle

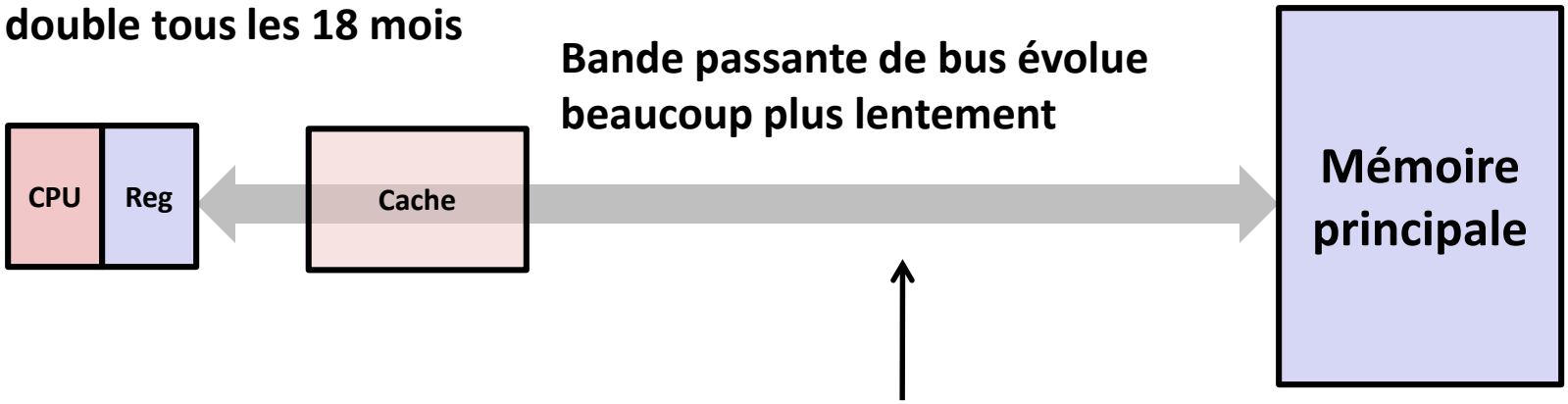
Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100 cycles

Problème: beaucoup d'attente mémoire !

Goulot d'étranglement : processeur - mémoire

La performance de processeur double tous les 18 mois

Bande passante de bus évolue beaucoup plus lentement



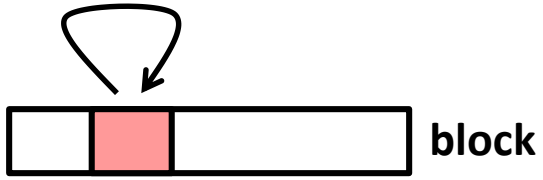
Core 2 Duo:
256 Bytes/cycle

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100 cycles

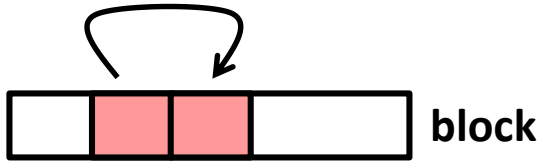
Solution: caches

Pourquoi cache marche ?

- **Localité: les programmes ont la tendance d'utiliser des données et des instructions avec des adresses voisines ou égales à celles qu'ils ont utilisé récemment**



- **Localité temporelle :**
  - Les items référencés récemment ont la tendance d'être re-référencés dans le futur proche



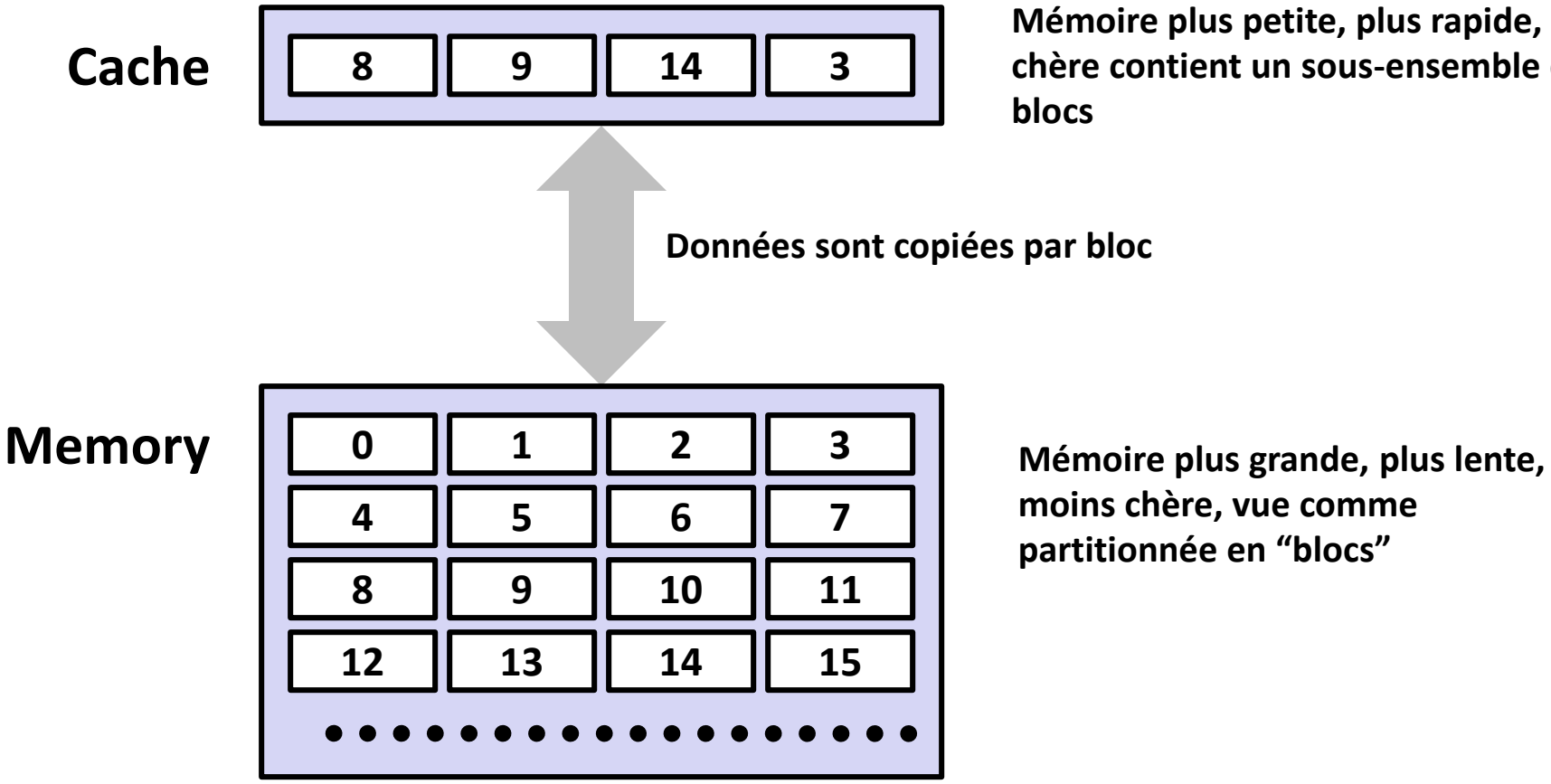
- **Localité spatiale :**
  - Les items avec des adresses voisines ont la tendance d'être référencés ensemble

Exemple de localité

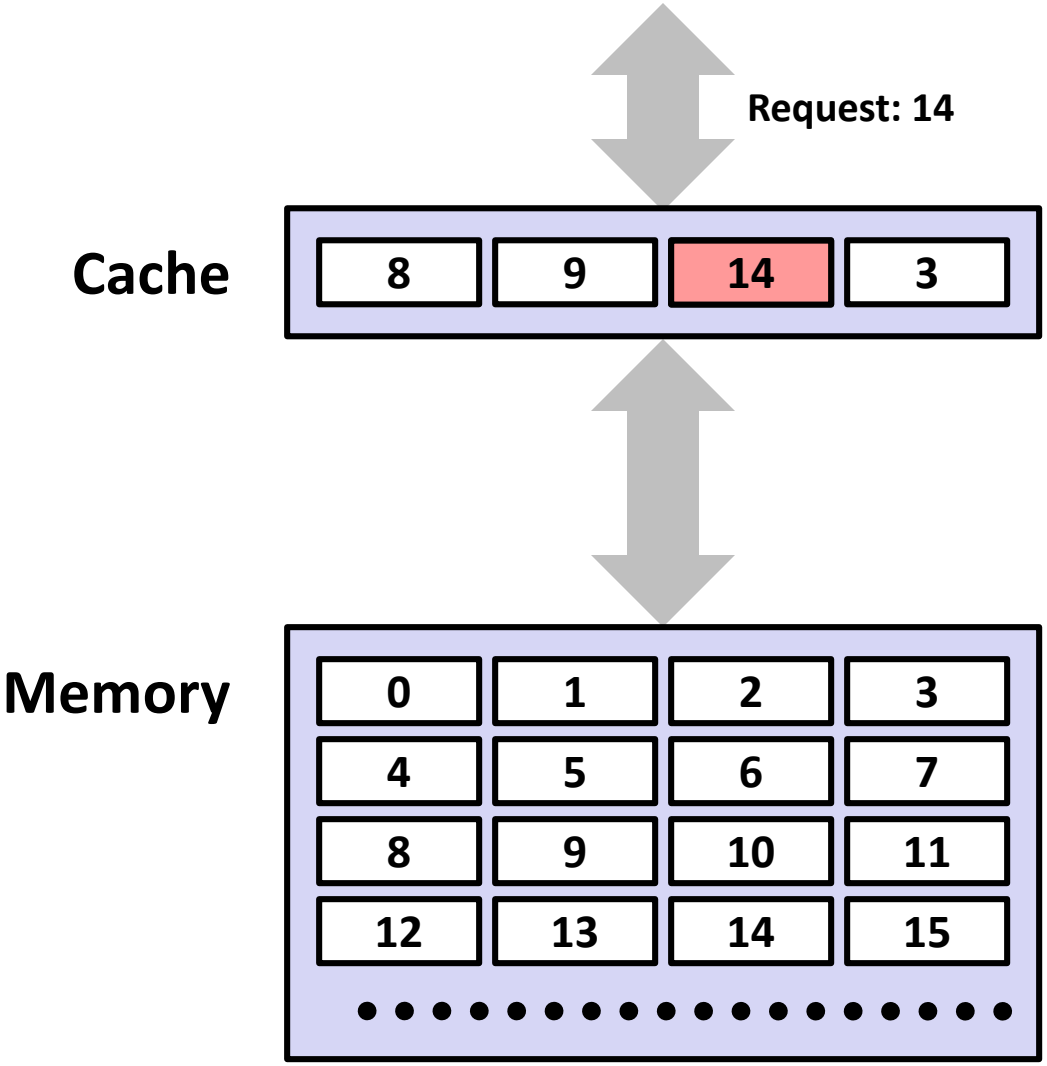
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

- Quel est le problème avec ce code ?
- Comment le corriger ?

Mécanisme général de cache



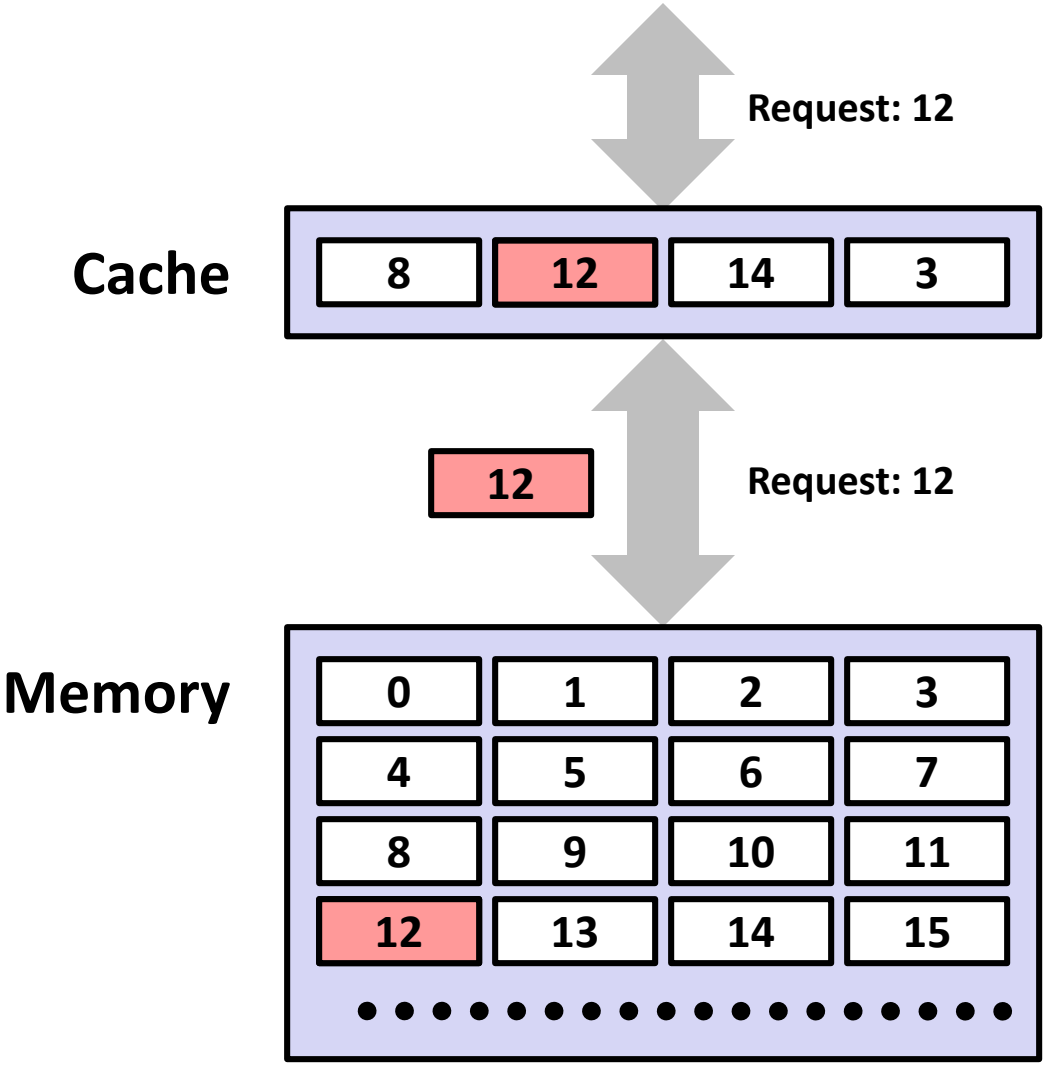
Concept général de cache: Hit



Besoin de données en bloc b

Bloc b est en cache :
Hit!

Concept général de cache: Miss



Besoin de données en bloc b

Bloc b n'est pas en cache: Miss!

Bloc b est récupéré de mémoire

Bloc b est stocké en cache

- Politique de placement : détermine où b va
- Politique de remplacement : détermine quel bloc à expluser

Coût de cache miss

- **Différence énorme entre hit et miss**

- **Croyez vous que 99% hits est 2 fois mieux que 97%?**
  - Considérer :
    Cache hit : 1 cycle
    Miss penalty : 100 cycles

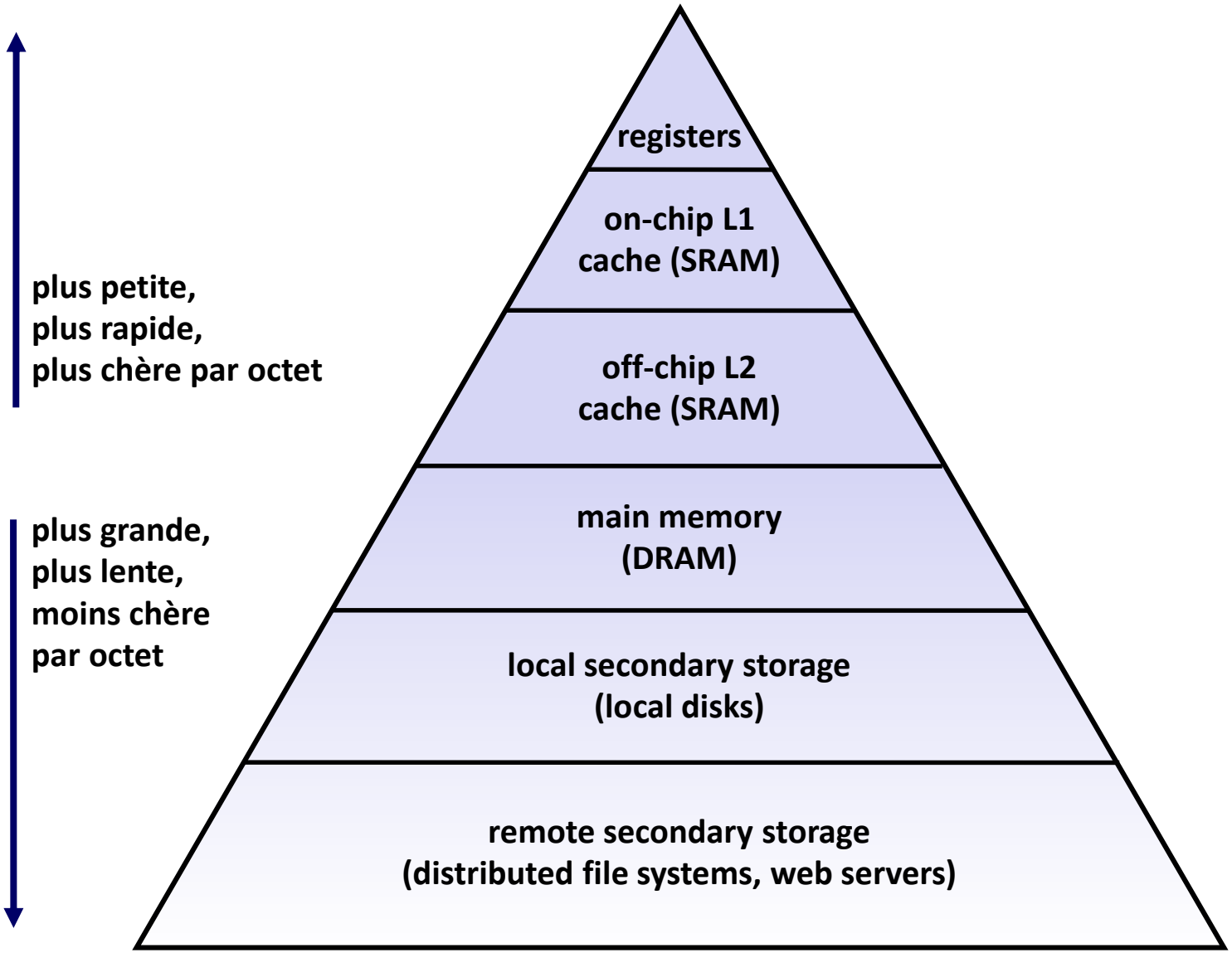
  - Temps d'accès moyen :
    - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

- **C'est pourquoi "miss rate" est utilisé au lieu de "hit rate"**

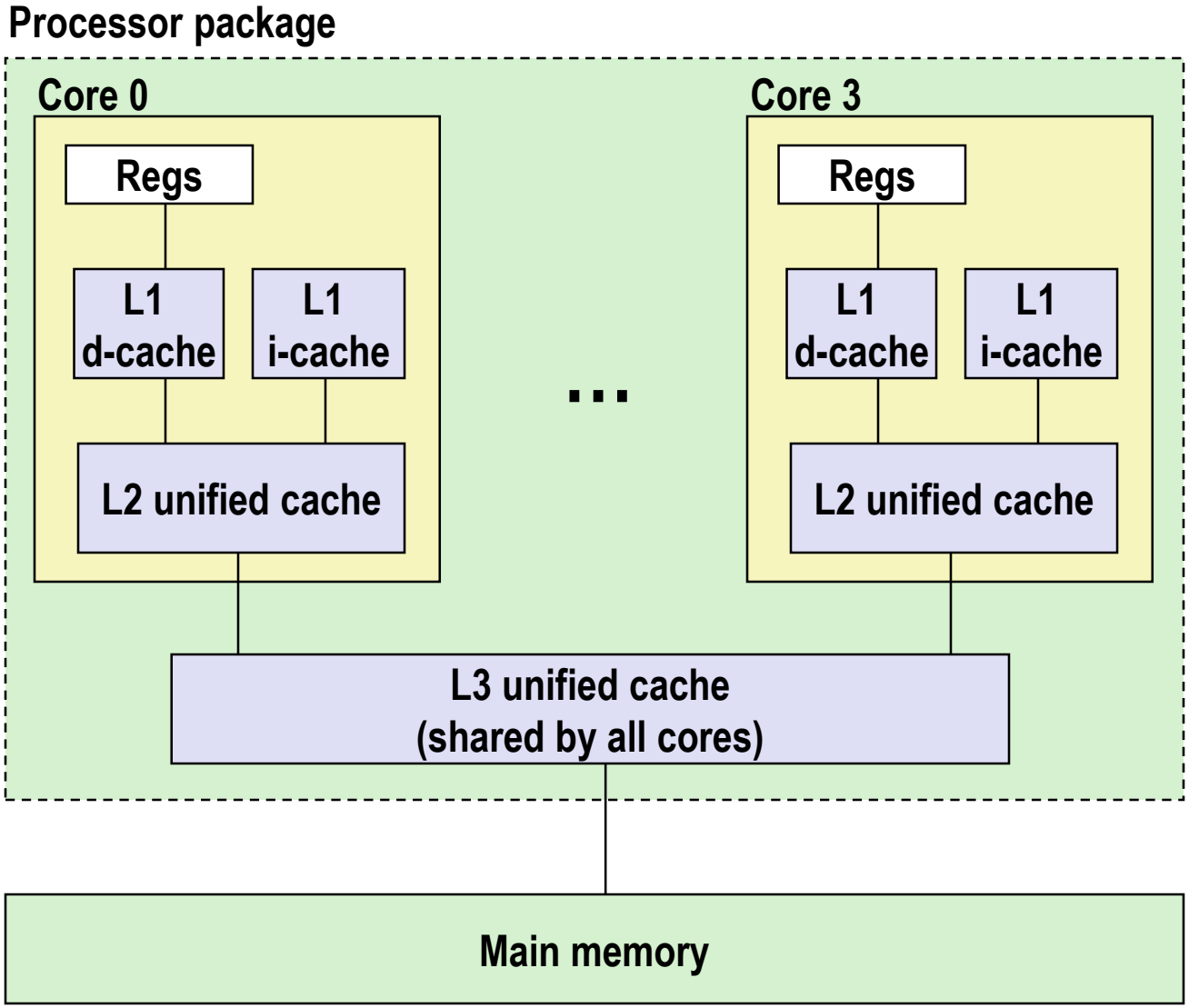
Cours 8: Mémoire et Caches

- ~~Principe de localité~~
- Hiérarchie de mémoire
- Organisation de cache
- Optimisations de programme avec cache

Hiérarchie de mémoire : exemple



Intel Core i7



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

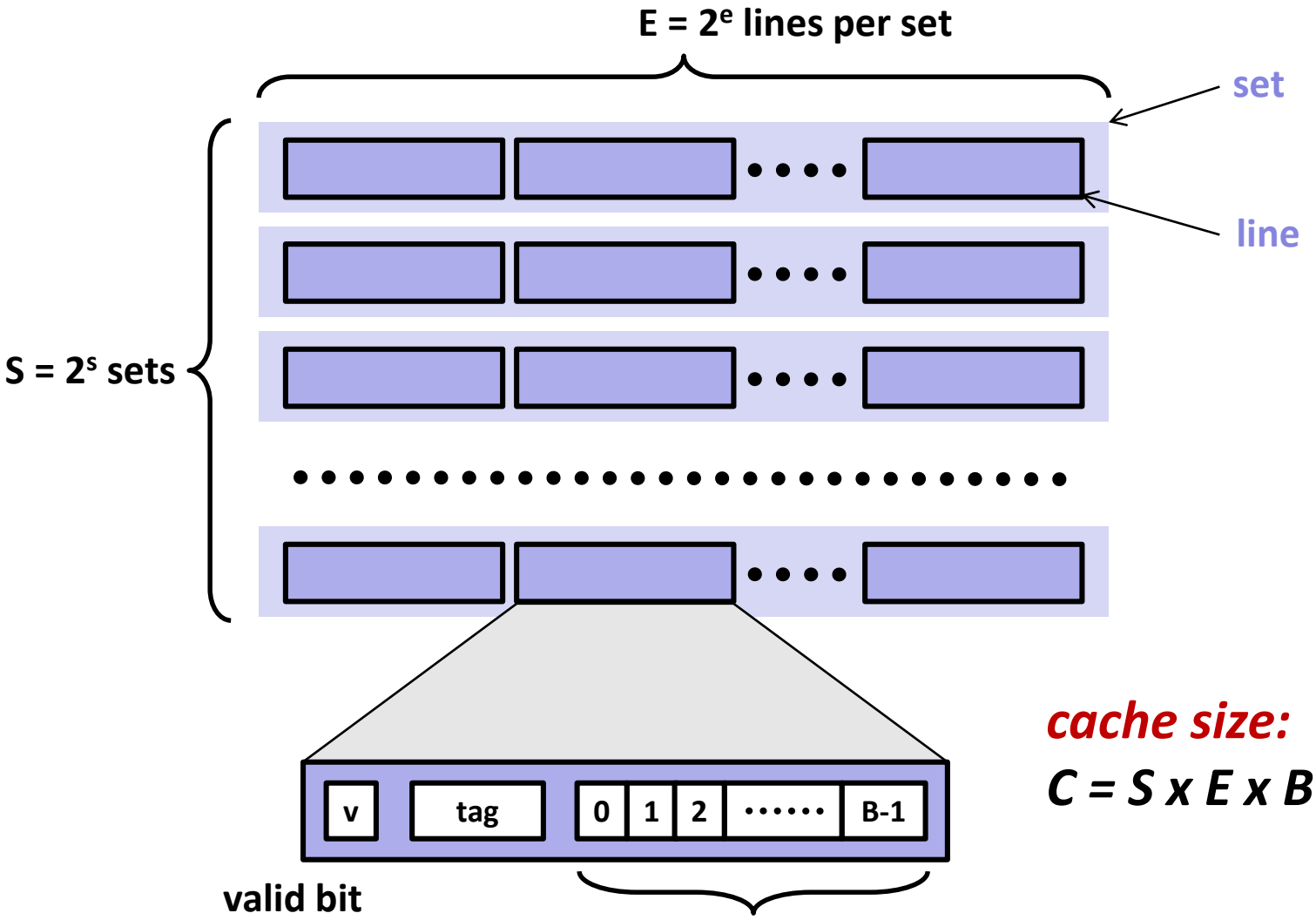
L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

Cours 8: Mémoire et Caches

- ~~Principe de localité~~
- ~~Hiérarchie de mémoire~~
- Organisation de cache
- Optimisations de programme avec cache

Organisation générale de cache (S, E, B)

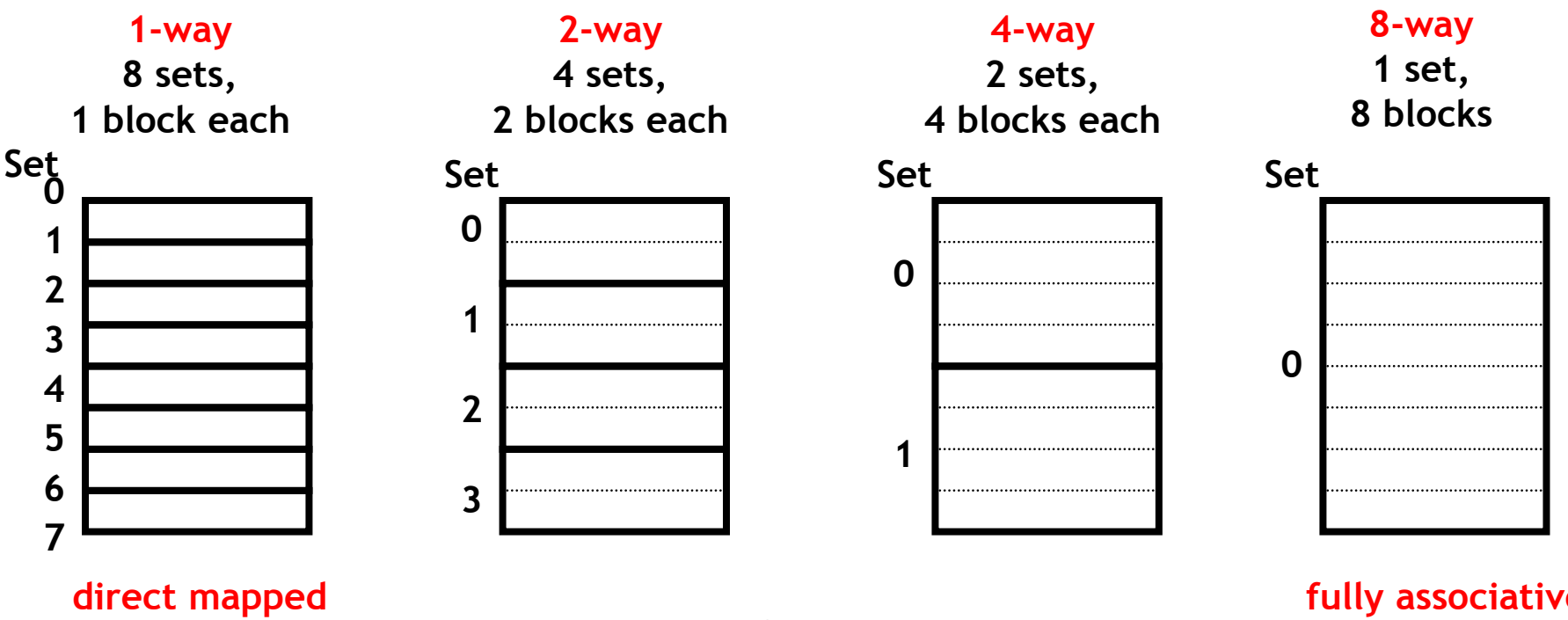


cache size:

C = S x E x B data bytes

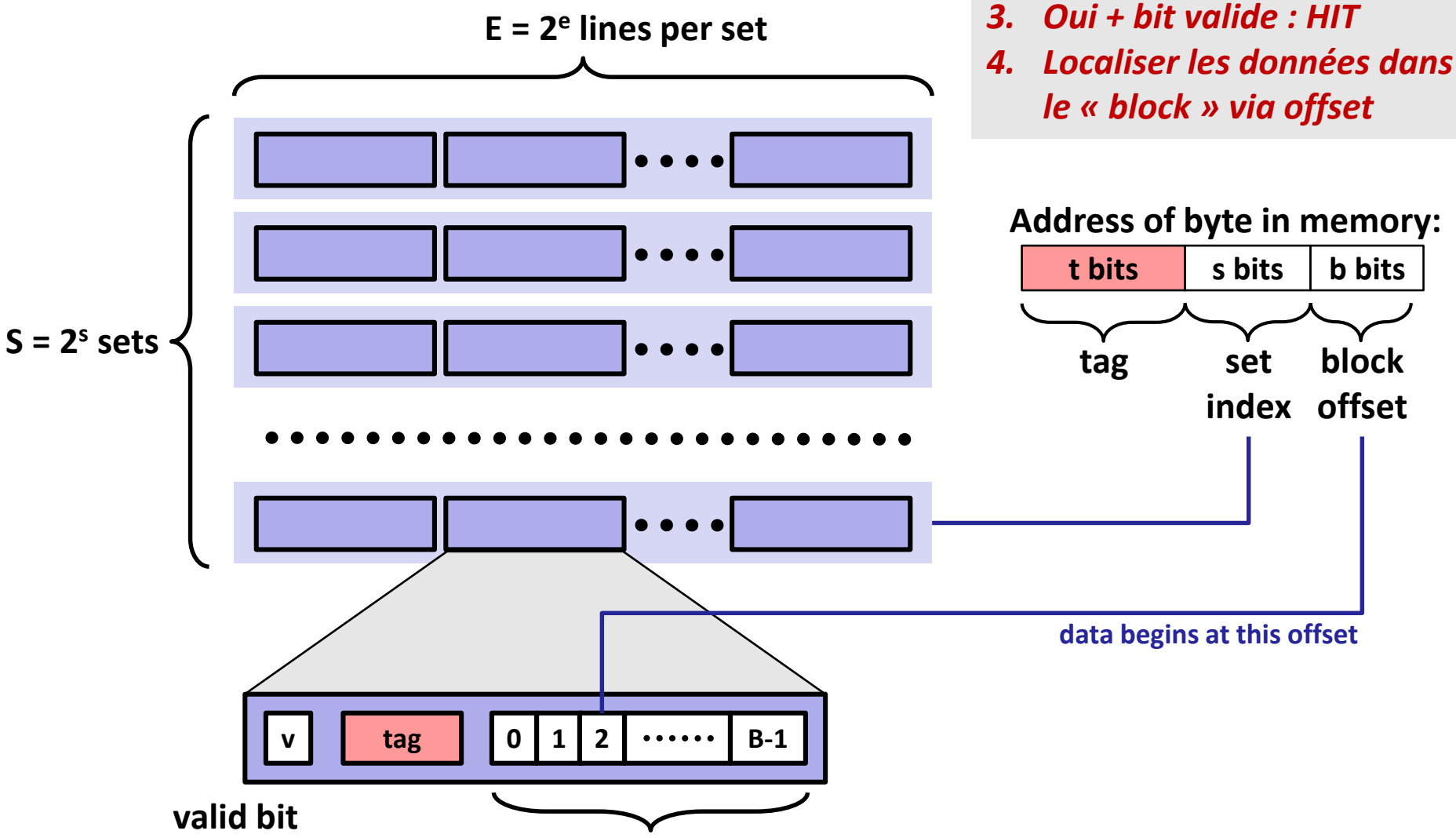
Différents cas

- **Direct-mapped cache : E = 1**
  - Une ligne par ensemble
- **E-way set-associative cache : 2 <= E < C/B**
  - E lignes par ensemble
- **Fully associative cache : E = C/B, S = 1**
  - Un seul ensemble avec C/B lignes



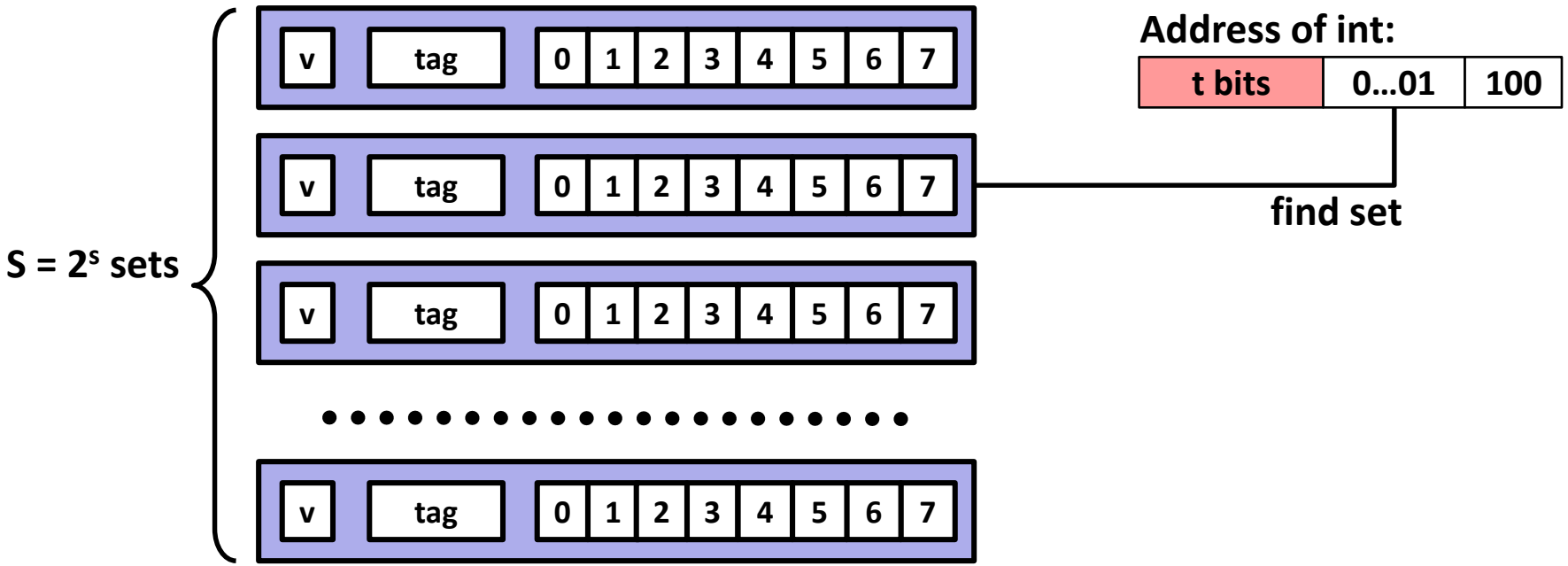
Cache Read

1. *Localiser « set »*
2. *Vérifier s'il y a une « line » qui a le même « tag »*
3. *Oui + bit valide : HIT*
4. *Localiser les données dans le « block » via offset*



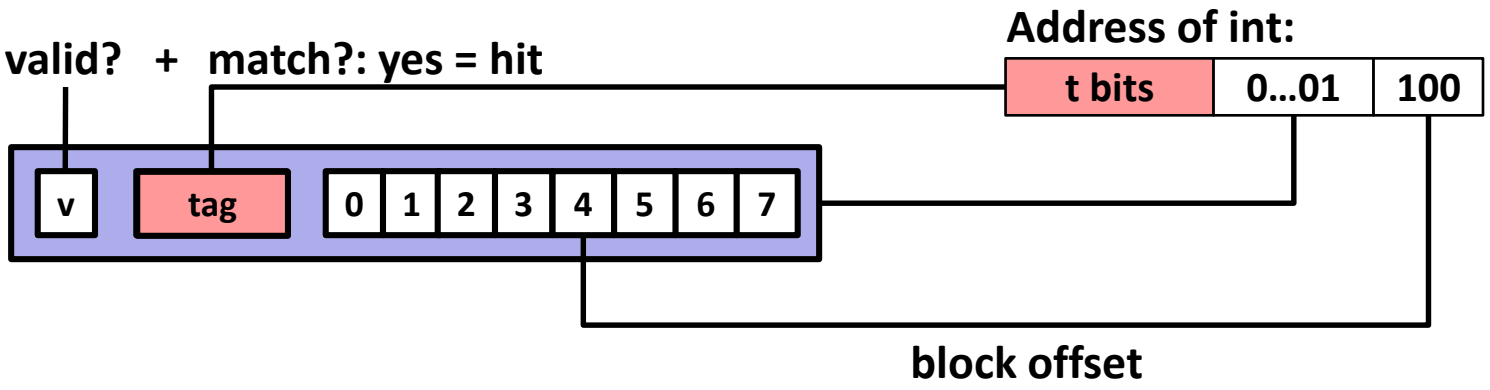
Direct-Mapped Cache : exemple

Direct-mapped: 1 ligne par ensemble
Taille de bloc de cache : 8 octets



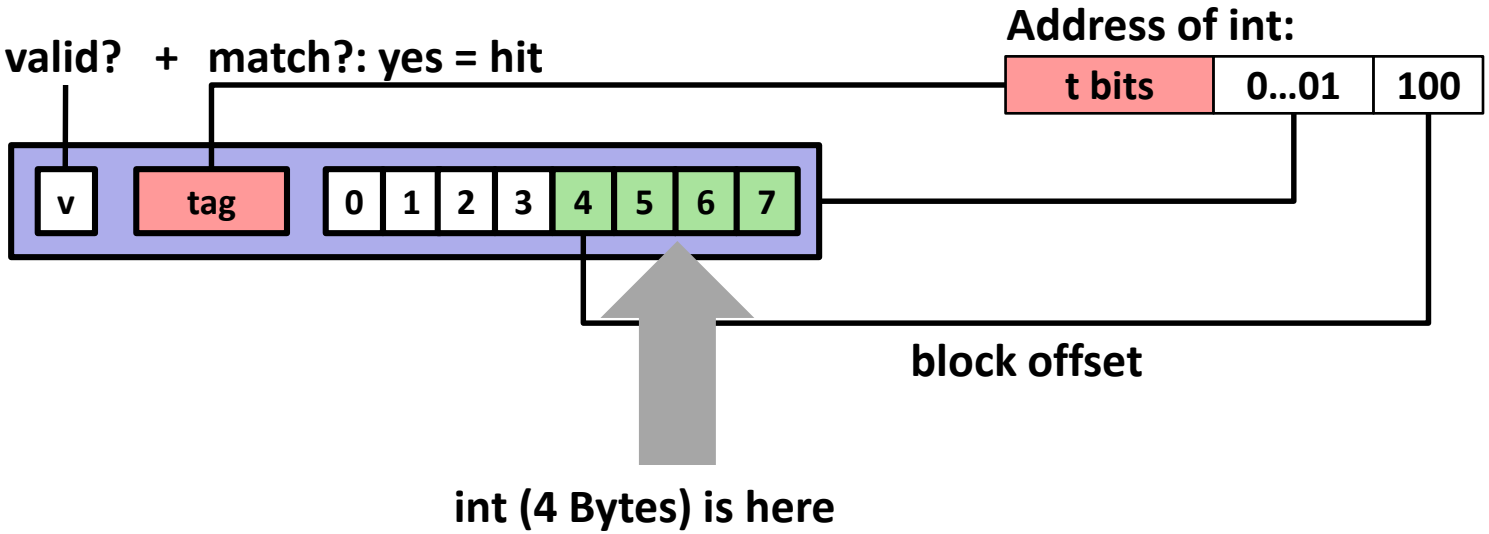
Direct-Mapped Cache : exemple

Direct-mapped: 1 ligne par ensemble
Taille de bloc de cache : 8 octets



Direct-Mapped Cache : exemple

Direct-mapped: 1 ligne par ensemble
Taille de bloc de cache : 8 octets

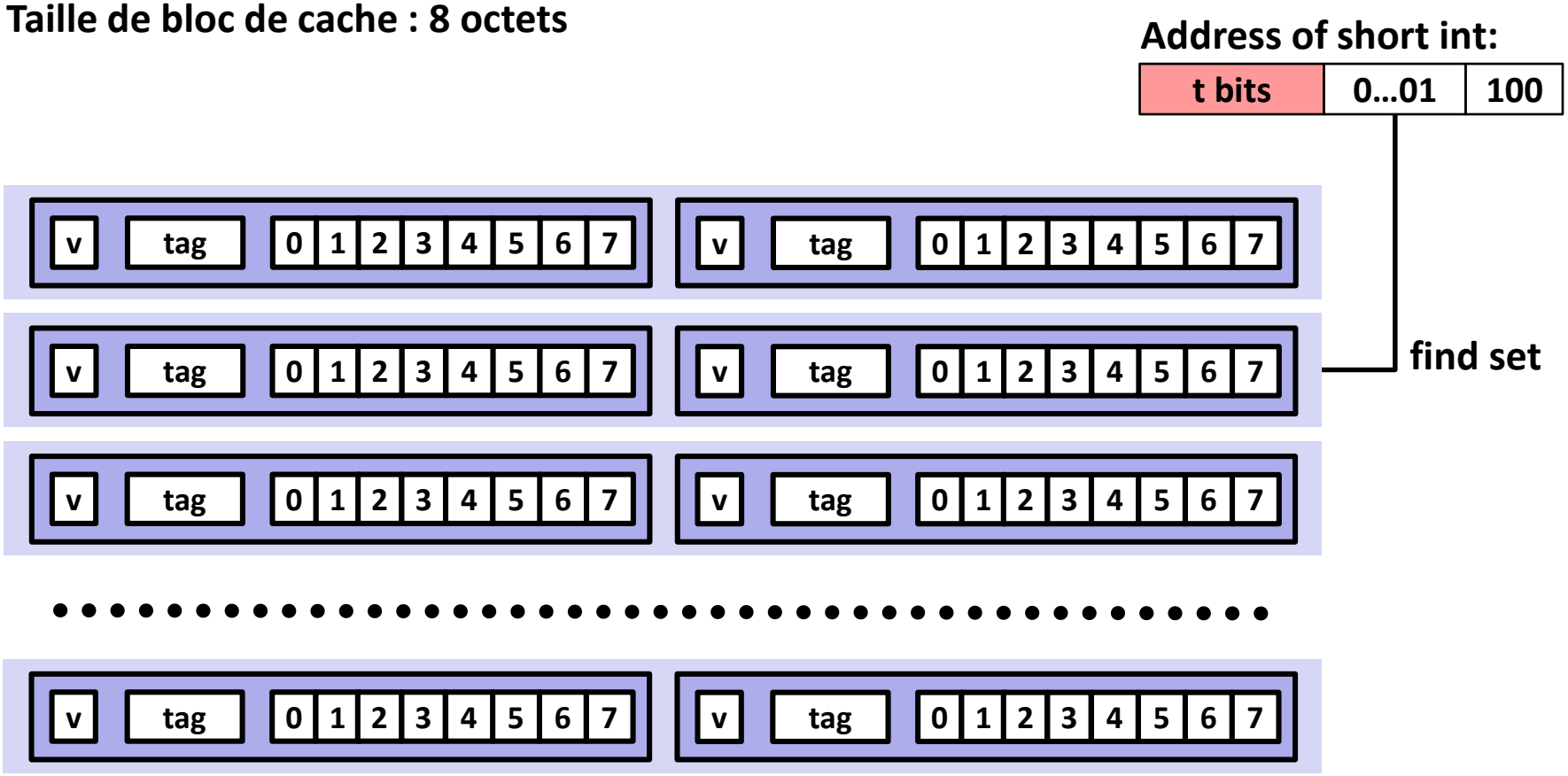


MISS: ancienne ligne est explusée et remplacée

E-way Set-Associative Cache : exemple

E = 2: deux lignes par ensemble

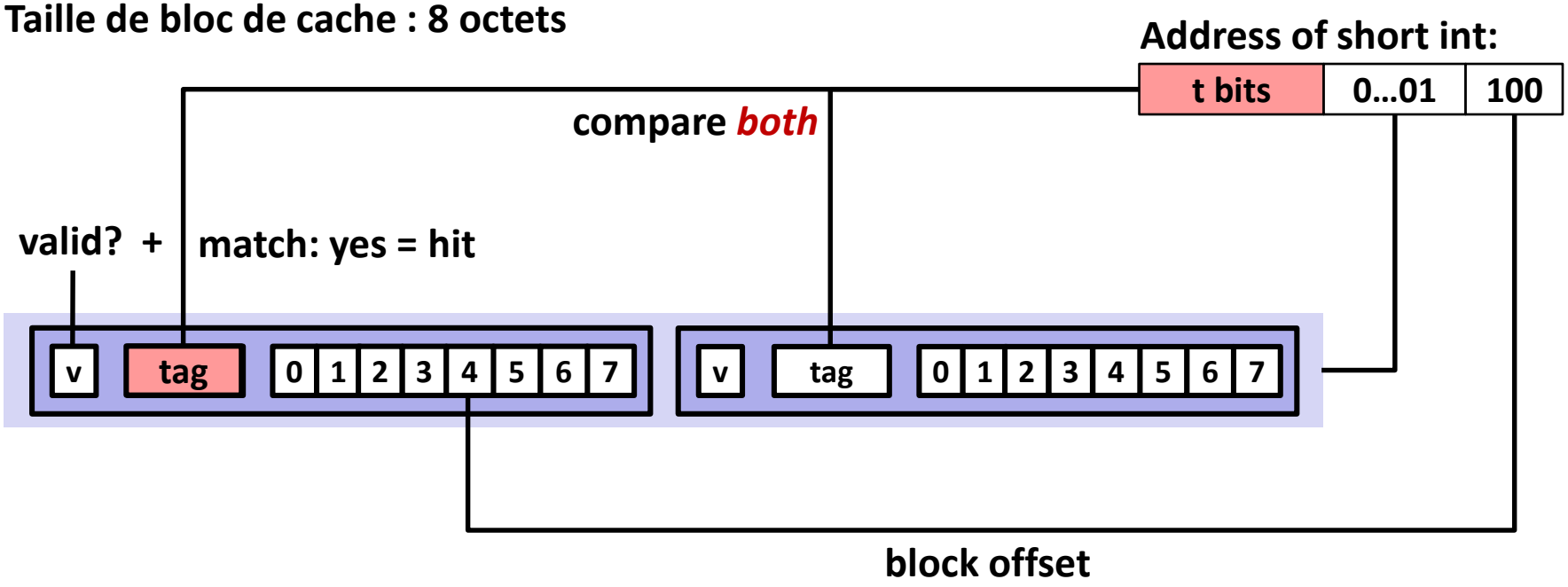
Taille de bloc de cache : 8 octets



E-way Set-Associative Cache : exemple

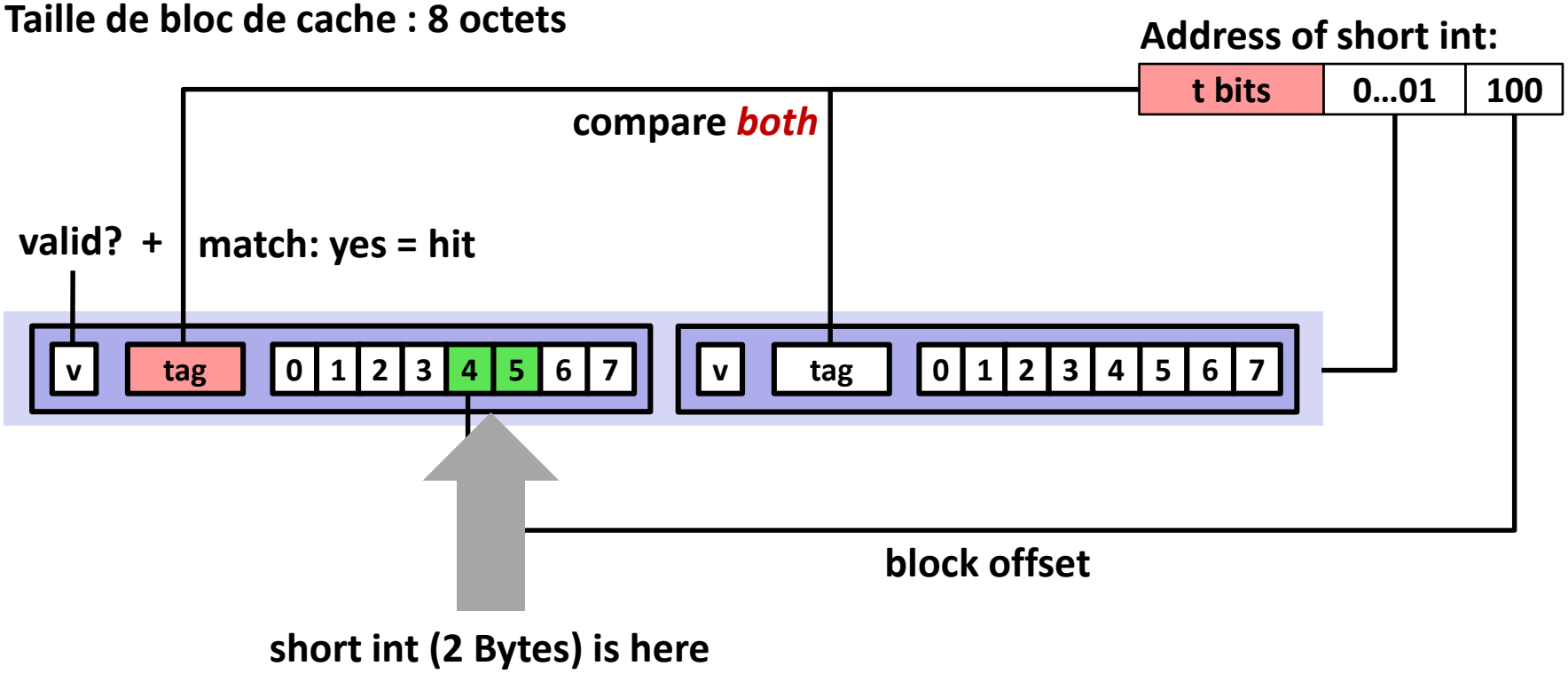
E = 2: deux lignes par ensemble

Taille de bloc de cache : 8 octets



E-way Set-Associative Cache : exemple

E = 2: deux lignes par ensemble
Taille de bloc de cache : 8 octets



MISS:

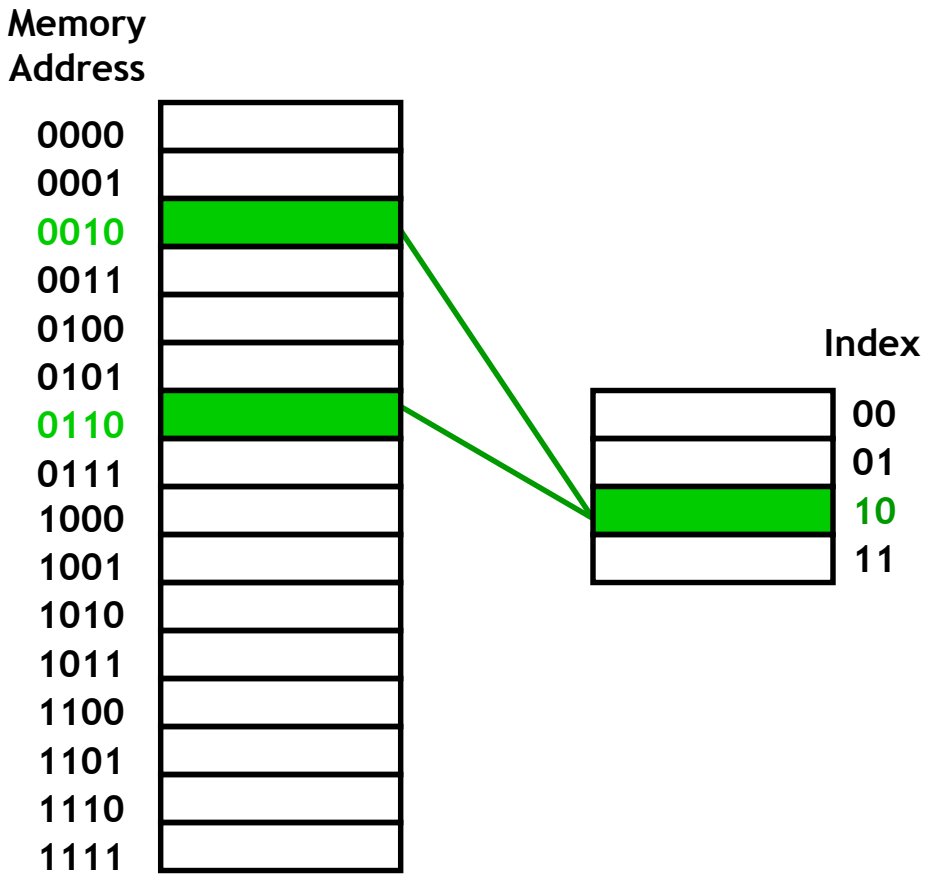
- Une ligne dans l'ensemble est choisie pour être supprimée
- Politique de remplacement : random, least recently used (LRU), …

Types de Cache Miss

- **Cold (compulsory) miss**
  - Apparaître au 1er accès d'un bloc
- **Conflict miss**
  - Apparaître quand le cache est assez grand mais des données multiples à référencer sont « mappées » au même endroit
- **Capacity miss**
  - Apparaître quand les données (working set) ne rentrent pas dans le cache

Conflict miss : “direct-mapped cache”

- Quand le programme utilise les adresses 2, 6, 2, 6, 2, …
- Thrashing !



Et pour les écritures (WRITE) ?

- **Des copies multiples existent:**
  - L1, L2, L3, mémoire principale
- **Le problème ?**

Et pour les écritures (WRITE) ?

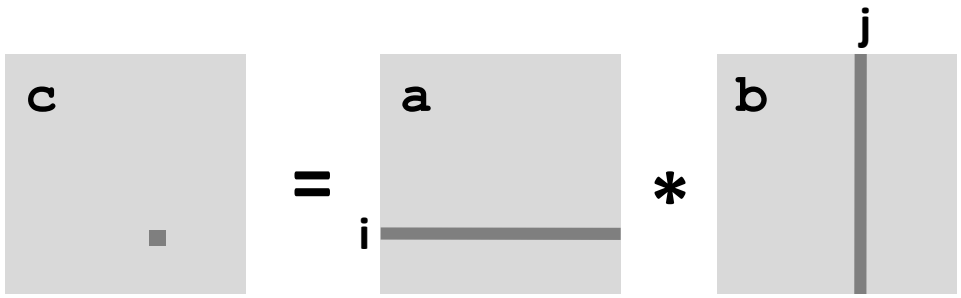
- **Des copies multiples existent:**
  - L1, L2, L3, mémoire principale
- **Quand il y a un write-hit?**
  - Write-through (écrire immédiatement à la mémoire)
  - Write-back (attarder l'écriture à la mémoire jusqu'à ce que la ligne est expulsée de cache)
    - Besoin d'un *dirty bit* pour indiquer si la ligne est différente de la mémoire ou pas
- **Quand il y a un write-miss?**
  - Write-allocate (charger le bloc dans le cache, puis mettre à jour la ligne)
    - OK si localité, mais coût de transfert chaque fois qu'il y a un cache miss
  - No-write-allocate (juste écrire immédiatement à la mémoire)
- **En réalité :**
  - Write-back + Write-allocate : souvent
  - Write-through + No-write-allocate : occasionnel

Cours 8: Mémoire et Caches

- ~~Principe de localité~~
- ~~Hiérarchie de mémoire~~
- ~~Organisation de cache~~
- **Optimisations de programme avec cache**
  - Réarranger les boucles
  - Blocking

Exemple: multiplication matricielle

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and 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][j] += a[i][k]*b[k][j];
}
```



```
/* Version 1 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;
    }
```

```
/* Version 2 jik */
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k]*b[k][j];
        c[i][j] += sum;
    }
```

```
/* Version 3 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;
    }
```

```
/* Version 4 kji */
for (k = 0; k < n; k++)
    for (j = 0; j < n; j++) {
        r = b[k][j];
        for (i = 0; i < n; i++)
            c[i][j] += a[i][k]*r;
    }
```

```
/* Version 5 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];
    }
```

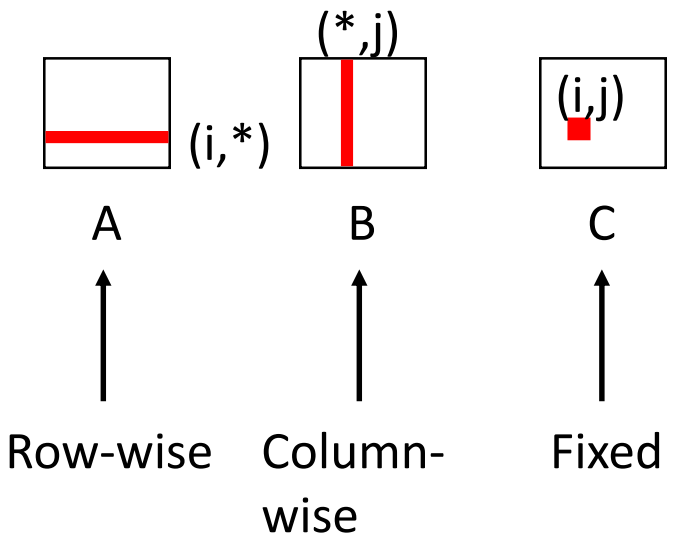
```
/* Version 6 ikj */
for (i = 0; i < n; i++)
    for (k = 0; k < n; k++) {
        r = a[i][k];
        for (j = 0; j < n; j++)
            c[i][j] += r*b[k][j];
    }
```

Analyse de Cache Miss

- **Hypothèses :**
  - Matrice de n x n « doubles » (8 octets)
  - Cache block = 32 bytes (4 doubles)
  - Cache size C << n

Matrix Multiplication (Version 1 ijk)

```
/* Version 1 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;
  }
}
```

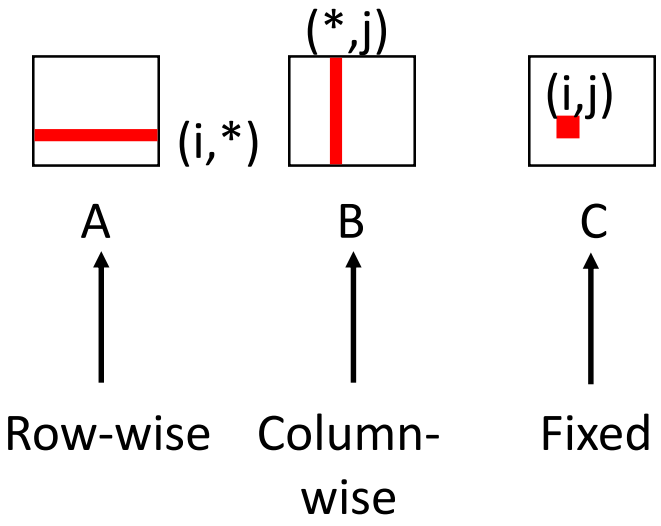


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (Version 2 jik)

```
/* Version 2 jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

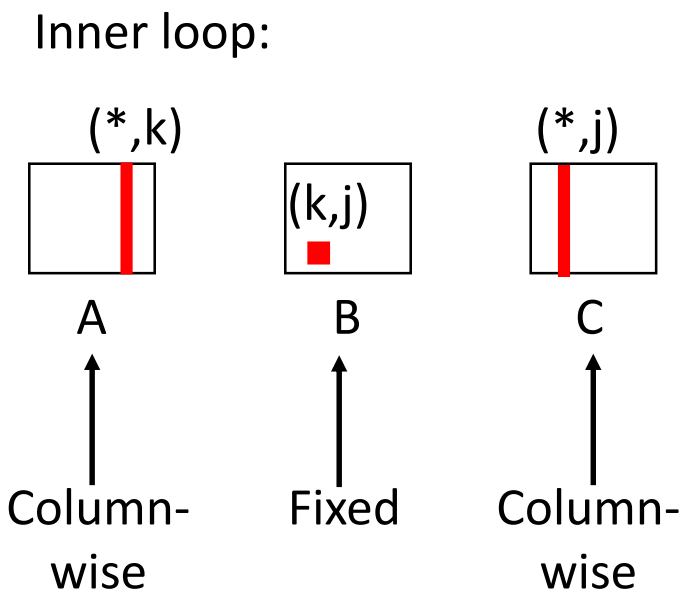


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (Version 3 jki)

```
/* Version 3 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;
  }
}
```

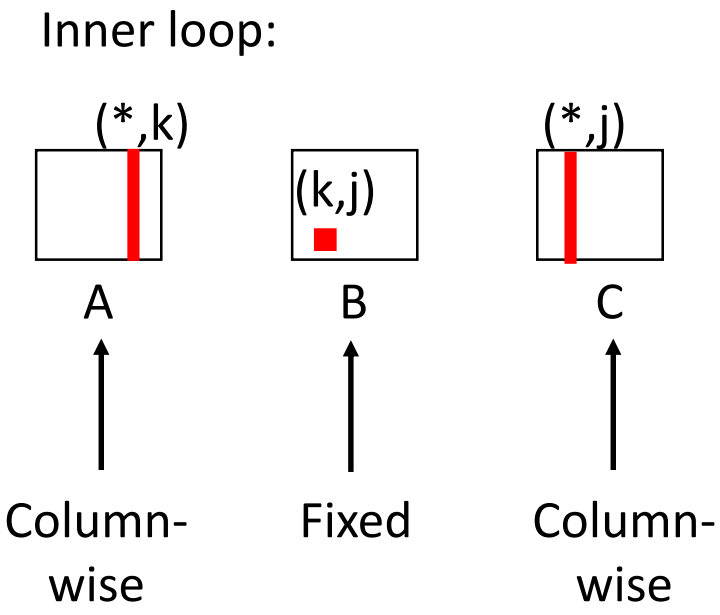


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (Version 4 kji)

```
/* Version 4 kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

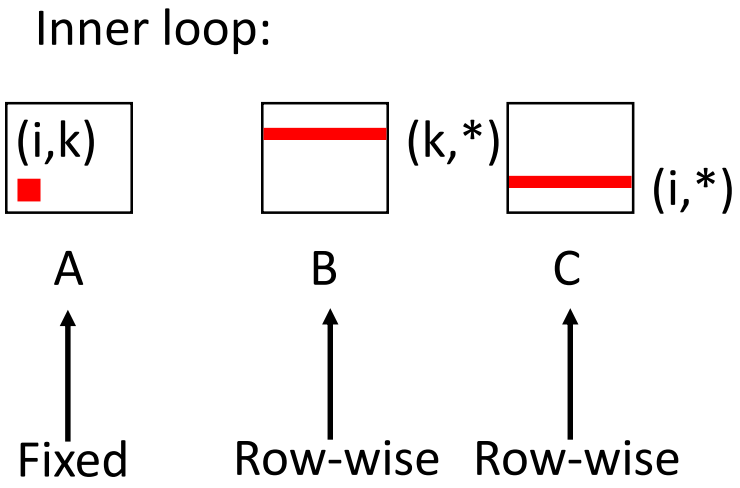


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (Version 5 kij)

```
/* Version 5 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];
  }
}
```

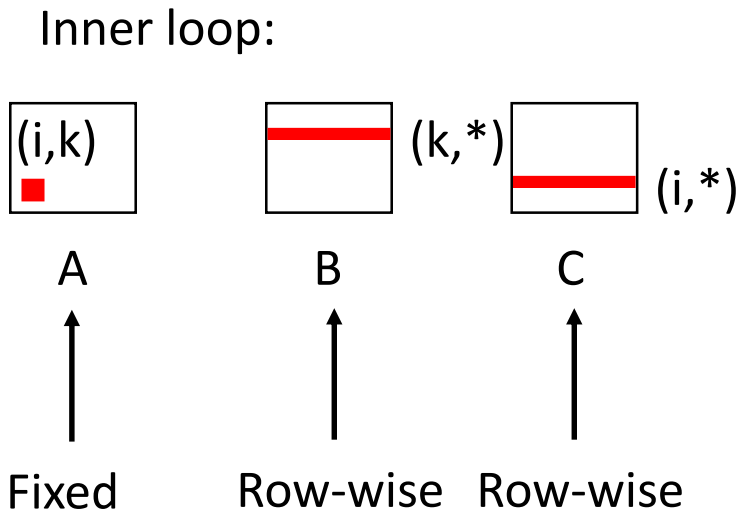


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (Version 6 ikj)

```
/* Version 6 ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Sommaire

```
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;
 }
}
```

Version 1 ijk (& Version 2 jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
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;
 }
}
```

Version 3 jki (& Version 4 kji):

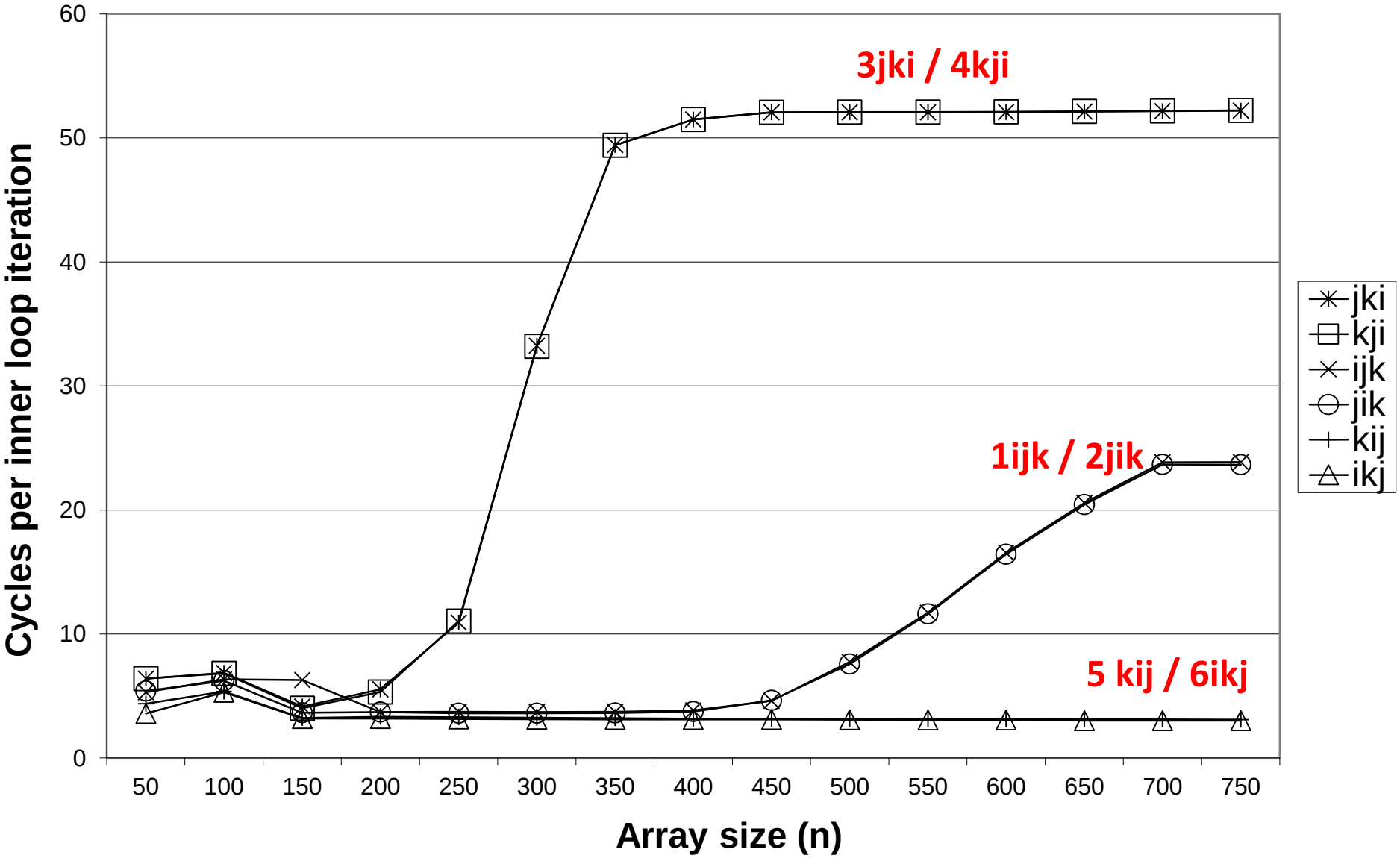
- 2 loads, 1 store
- misses/iter = **2.0**

```
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];
 }
}
```

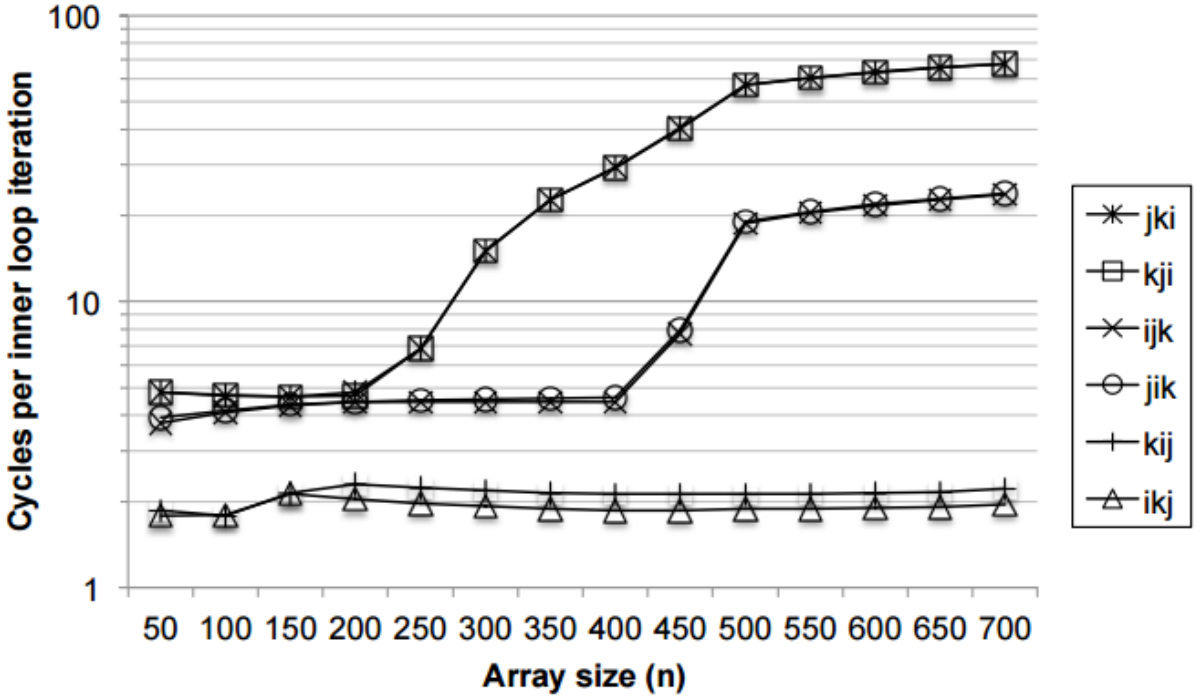
Version 5 kij (& Version 6 ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

Intel Core i7 : performance



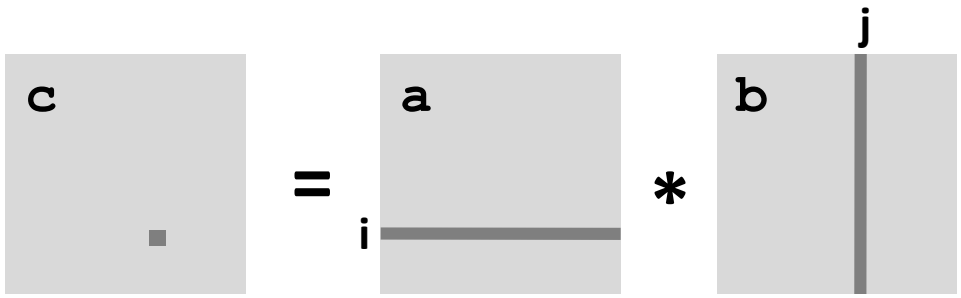
Intel Core i7 : performance



- n grand : 40x !
- même nombre de références mémoires + nombre de misses => même performance
- miss rate est un meilleur indicateur, par rapport au nombre de références mémoires
- n grand : performance constante pour versions 5 + 6 => Intel prefetching hardware !

Exemple: multiplication matricielle

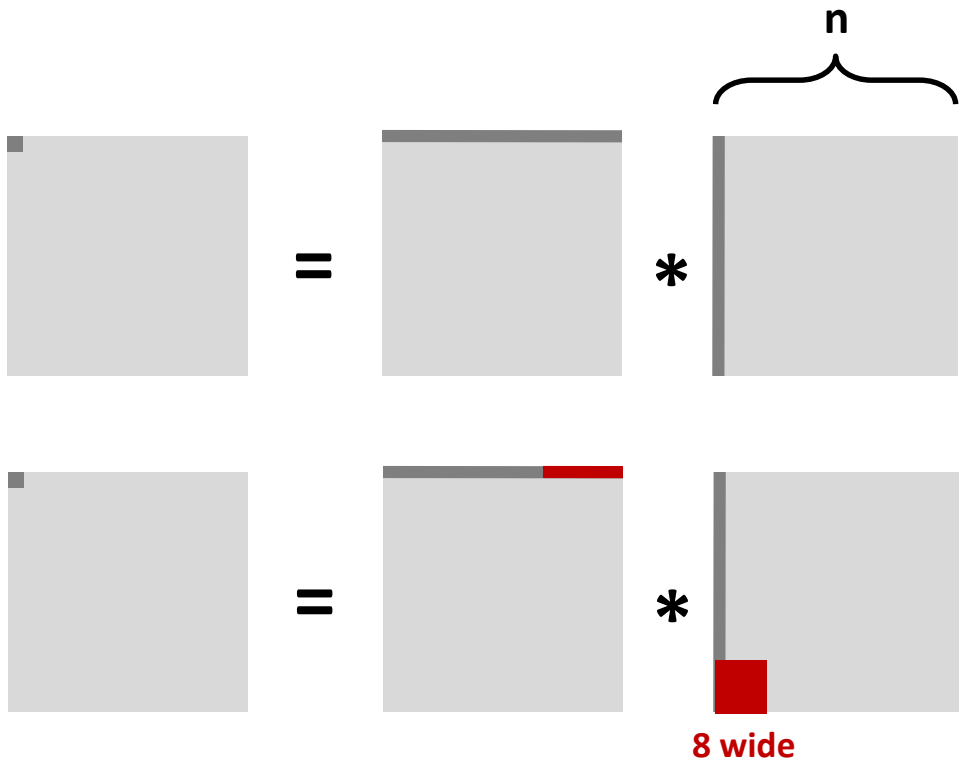
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and 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][j] += a[i][k]*b[k][j];
}
```



Analyse de Cache Miss

- **Hypothèses :**
  - Éléments de type « double »
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n

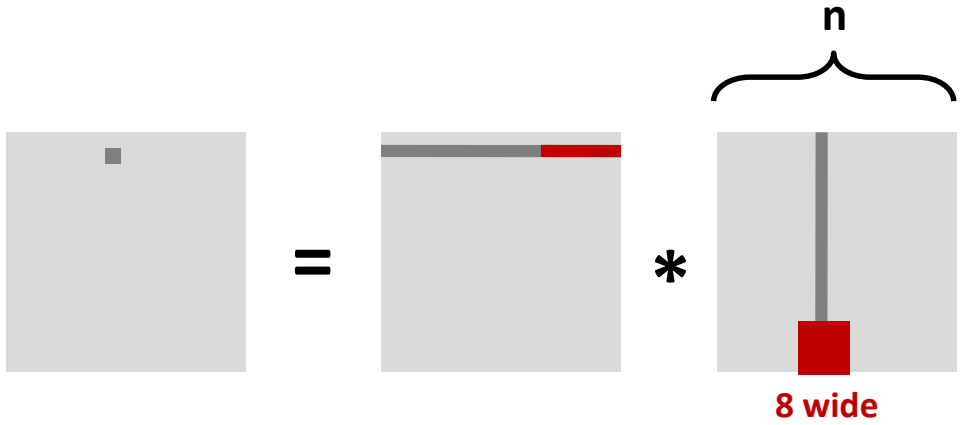
- **1ère itération:**
  - n/8 + n = 9n/8 misses (ne compte pas C)
  - Cache après



Analyse de Cache Miss

- **Hypothèses :**
  - Éléments de type « double »
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n

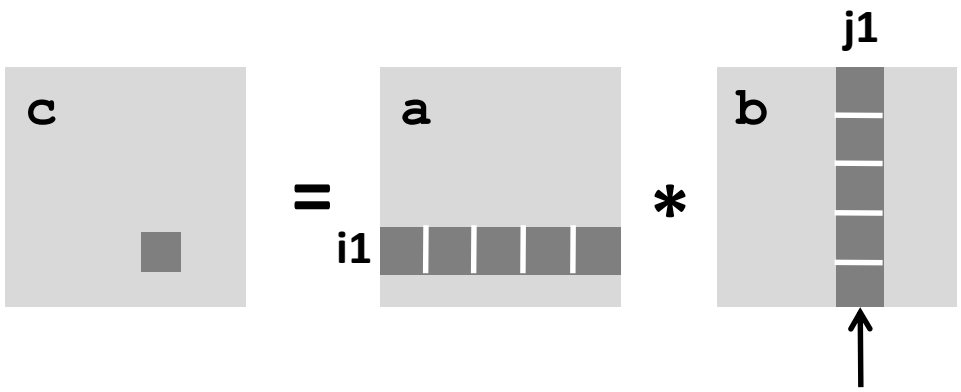
- **Autres itérations:**
  - n/8 + n = 9n/8 misses (ne compte pas C)



- **Nombre total de miss:**
  - $9n/8 * n^2 = (9/8) * n^3$

Multiplication matricielle avec blocs

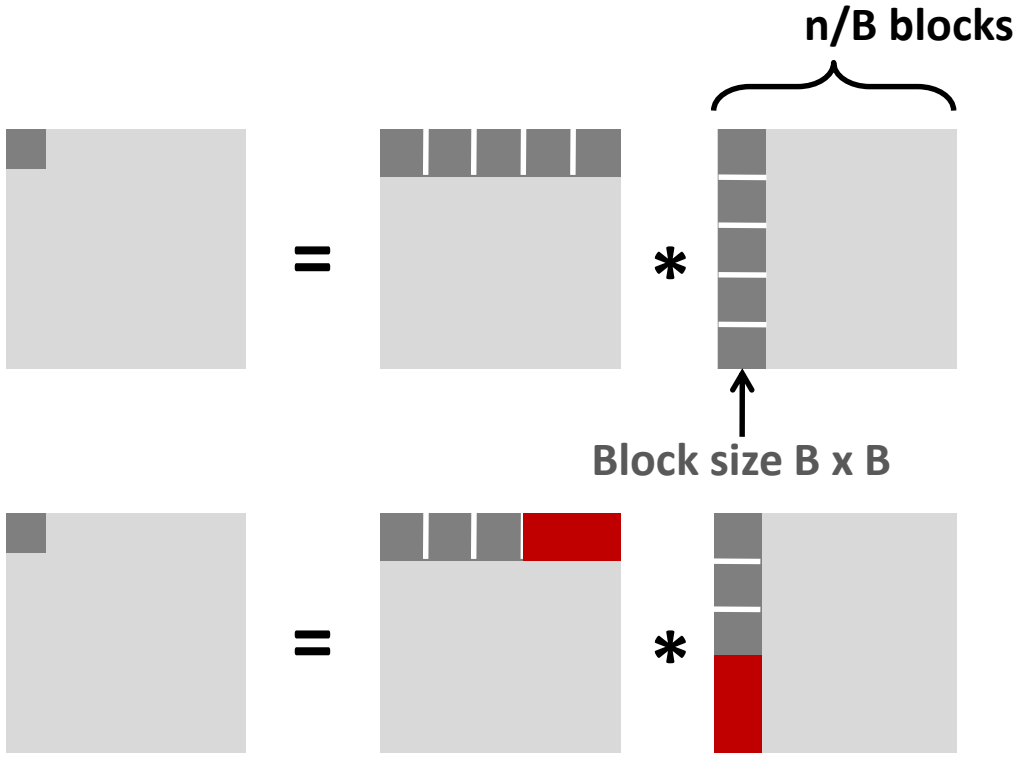
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and 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)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1][j1] += a[i1][k1]*b[k1][j1];
}
```



Analyse de Cache Miss

- **Hypothèses :**
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n
  - Trois blocks ■ rentrent en cache: $3B^2 < C$

- **1ère itération:**
  - $B^2/8$ miss pour chaque bloc
  - $2n/B * B^2/8 = nB/4$
    (ne compte pas C)



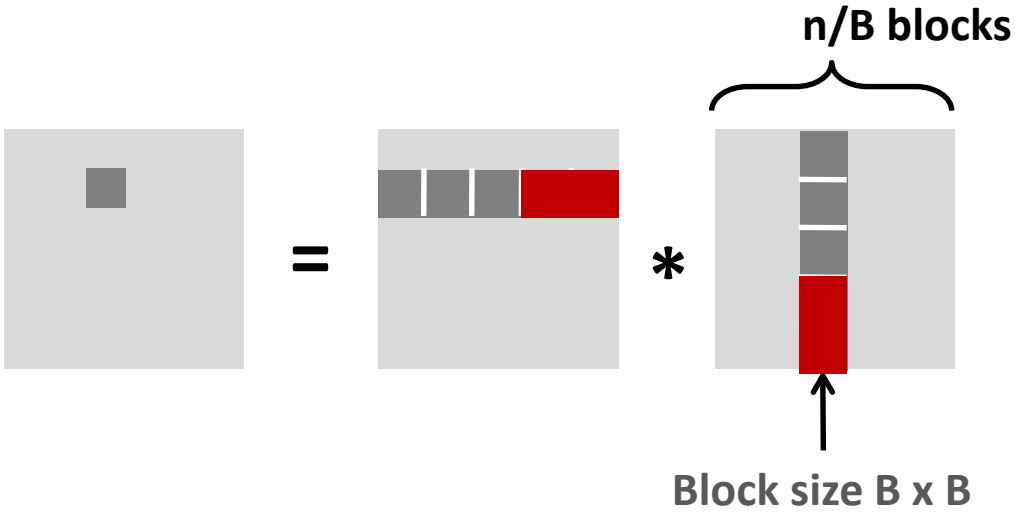
  - Cache après

Analyse de Cache Miss

- **Hypothèses :**
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n
  - Trois blocks ■ rentrent en cache: $3B^2 < C$

- **Autres itérations:**
  - $2n/B * B^2/8 = nB/4$

- **Nombre total de miss:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$



Comparaison

- Pas de blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- Si B = 8 : 36x
- Si B = 16 : 72x

- Bloc de taille plus grande, mais $3B^2 < C$!

Cache-Friendly Code

- **Programmeur peut optimiser le code par rapport au cache :**
  - Comment les structures de données sont organisées ?
  - Comment les données sont accédées ?
    - Structure de la boucle imbriquée
    - Blocking
- **Tous les systèmes favorisent “cache-friendly code”**
  - La performance optimale dépend beaucoup de plateforme
    - Taille de cache, taille de ligne, associativités, etc.
  - Règles générales
    - Garder le « working set » raisonnablement petit (localité temporelle)
    - Utiliser des pas petits (localité spatiale)
    - Concentrer au code de la boucle la plus intérieure

The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip