

# Fonctionnement et optimisations des mémoires cache

Nathalie Drach

[Nathalie.Drach@lip6.fr](mailto:Nathalie.Drach@lip6.fr)

Equipe ASIM

Laboratoire d'Informatique de Paris 6 (LIP6)

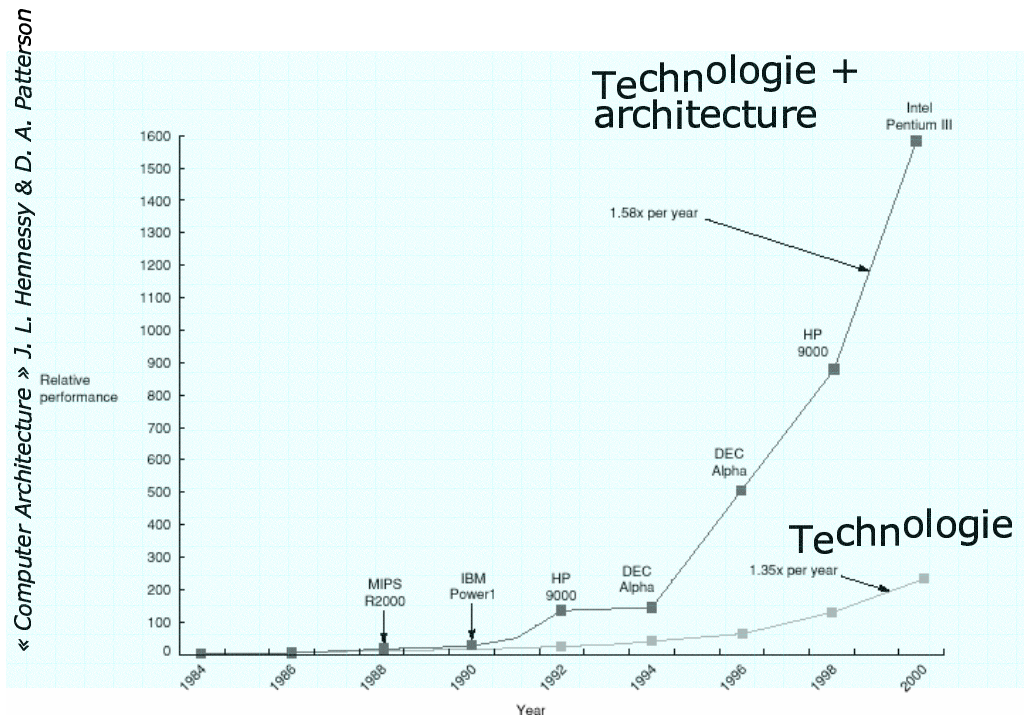
Université de Pierre et Marie Curie



# Apport de l'architecture

Augmentation des performances des processeurs : combinaison entre technologie et (micro)architecture.

- Evolutions technologiques :
  - Augmentation de la fréquence.
  - Augmentation du nombre de transistors.
- Evolutions en architecture :
  - Traduire l'augmentation des transistors en augmentation de performance.

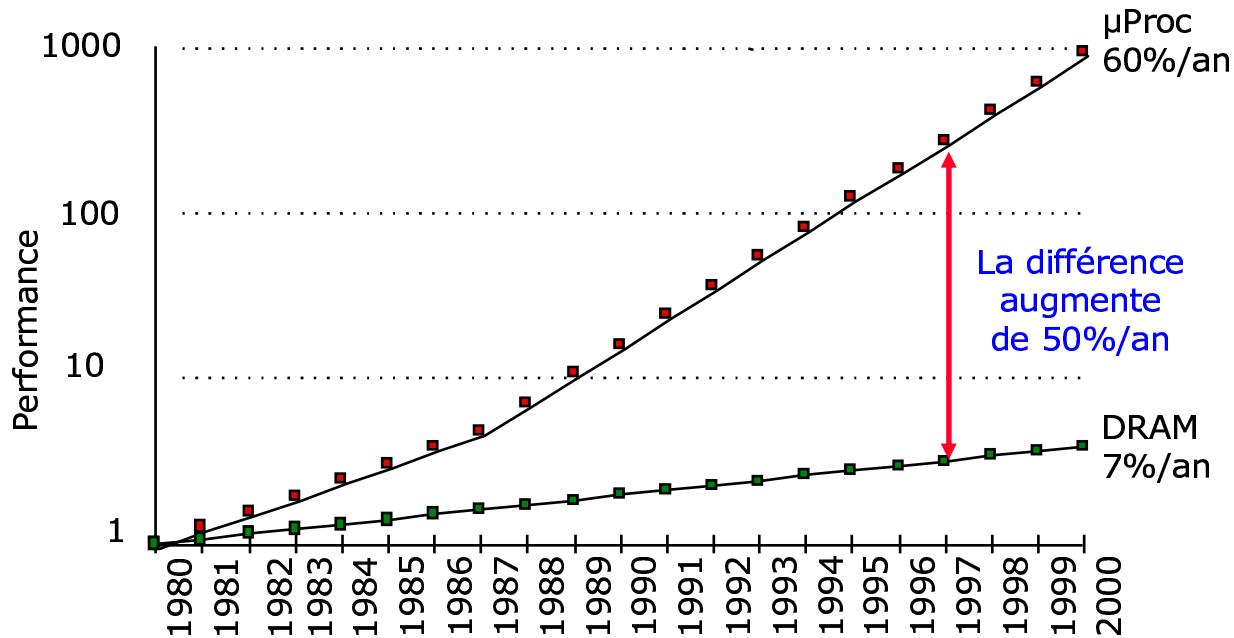


SIA RoadMap

Year	1999	2002	2005	2008	2011	2014
Feature size (nm)	180	130	100	70	50	35
Logic trans/cm <sup>2</sup>	6.2M	18M	39M	84M	180M	390M
Cost/trans (mc)	1.735	.580	.255	.110	.049	.022
#pads/chip	1867	2553	3492	4776	6532	8935
Clock (MHz)	1250	2100	3500	6000	10000	16900
Chip size (mm <sup>2</sup> )	340	430	520	620	750	900
Wiring levels	6-7	7	7-8	8-9	9	10
Power supply (V)	1.8	1.5	1.2	0.9	0.6	0.5
High-perf pow (W)	90	130	160	170	175	183
Battery pow (W)	1.4	2	2.4	2.8	3.2	3.7

“Prévisions” de la SIA (semiconductor industries association, <http://www.semichips.org/>) :

# Problème de la latence mémoire



## Problématique :

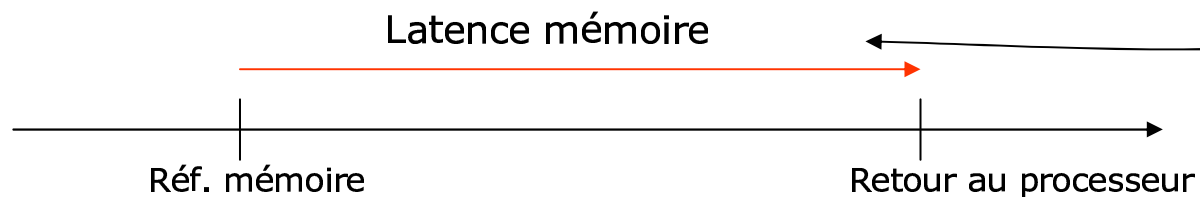
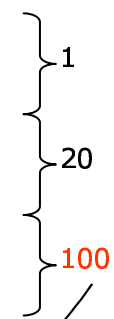
temps de cycle processeur  $\ll$  temps d'accès mémoire.

Requête mémoire du processeur = processeur inactif plusieurs cycles.

1986 : temps de cycle du processeur  $\sim 120$  ns  
temps d'accès à la mémoire  $\sim 140$  ns

1996 : temps de cycle du processeur  $\sim 4$  ns  
temps d'accès à la mémoire  $\sim 60$  ns

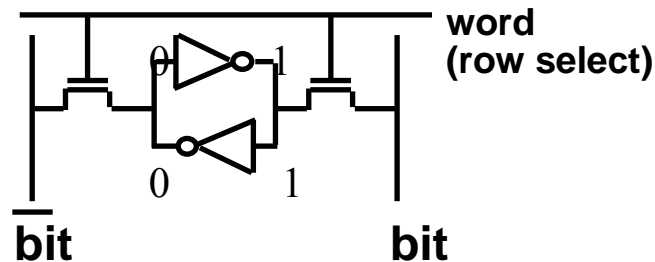
2002 : temps de cycle du processeur  $\sim 0.6$  ns  
temps d'accès à la mémoire  $\sim 50$  ns



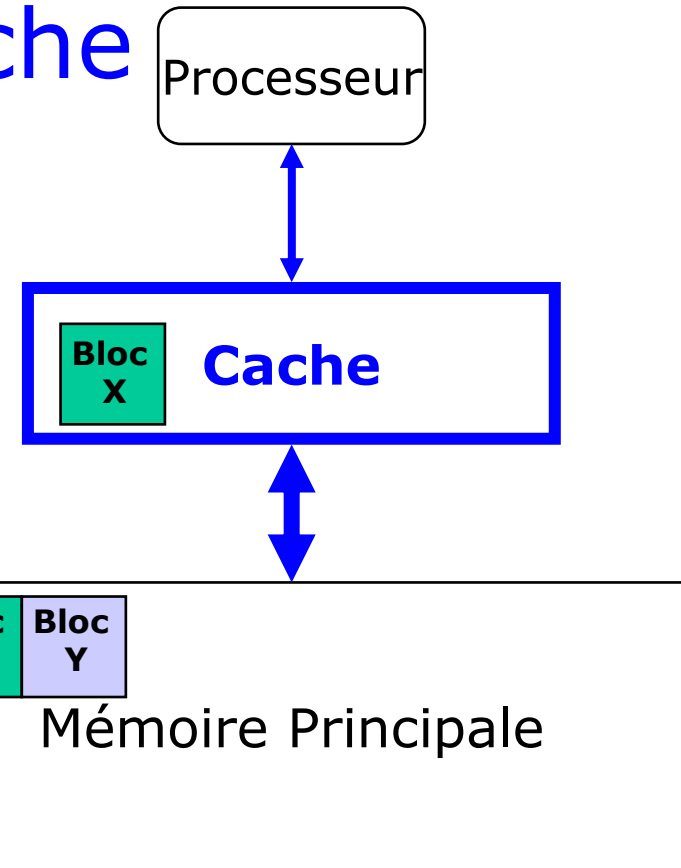
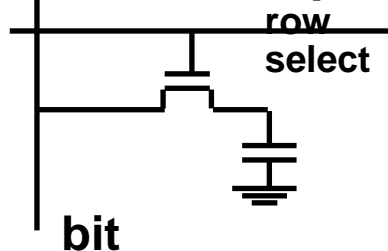
# Masquer la latence mémoire : Mémoire Cache

- Mémoire rapide (SRAM), mais petite (coût) : entre 1 et 3 cycles (accès éventuellement pipeline)

6-Transistor SRAM Cell

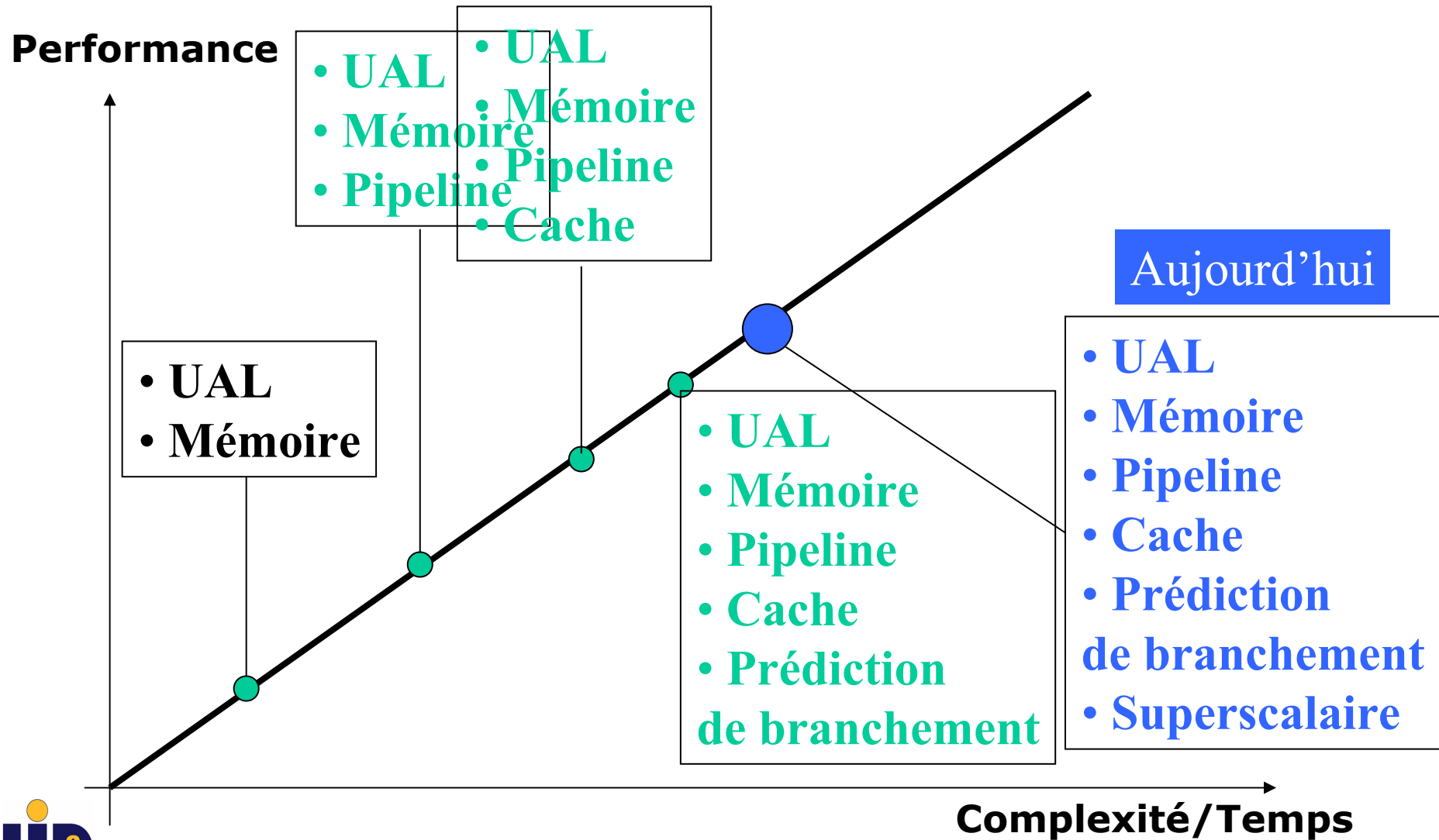


1-Transistor Memory Cell (DRAM)



Technologie mémoire	Temps accès typique	\$ par GB (2004)
SRAM	0.5 – 5 ns	\$4K-\$10K
DRAM	50 – 70 ns	\$100-\$200
Disque magnétique	5e6 – 20e6 ns	\$0.50-\$2

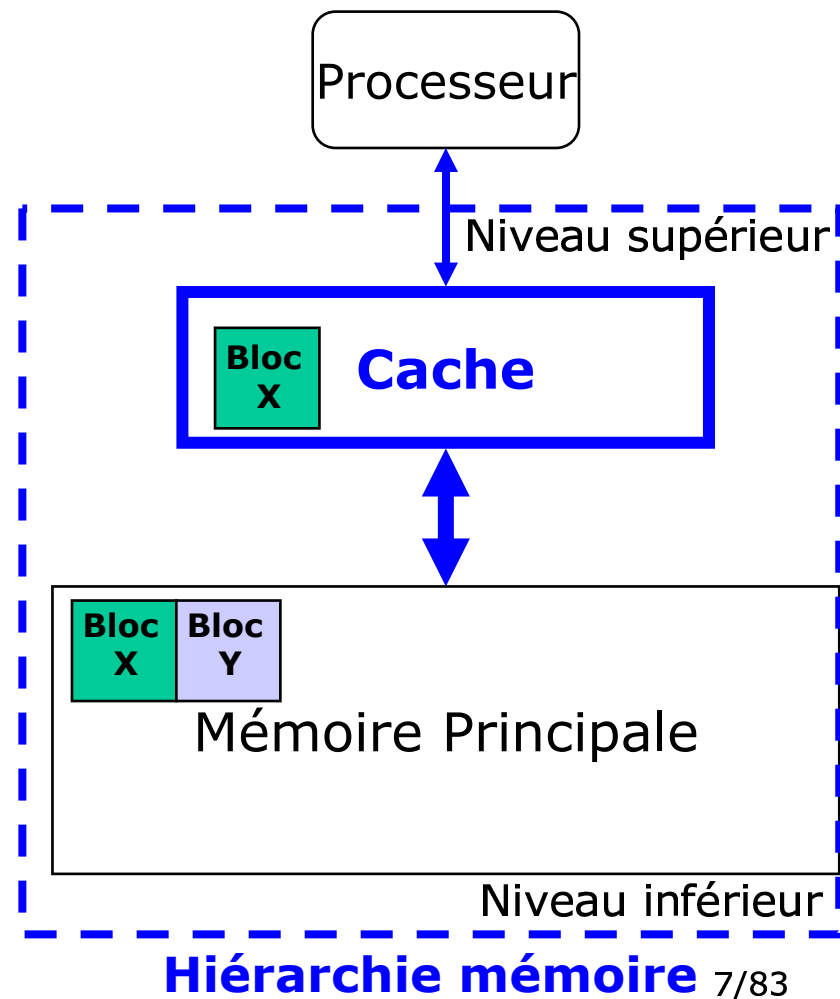
# Evolution architecture haute-performance



# Fonctionnement des mémoires cache

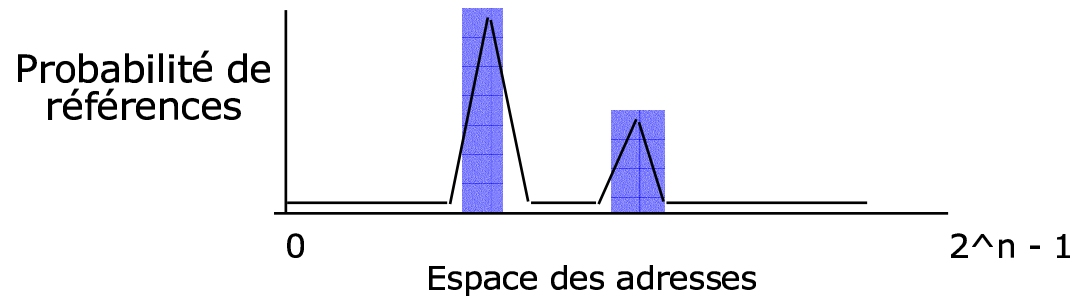
# Comment ça marche ?

- Le processeur envoie ses requêtes mémoire au cache :
  - donnée dans le cache : **succès** (*hit*)
  - donnée hors du cache : **échec** (*miss*)
- Temps de succès  $\ll$  pénalité d'échecs.
- Temps de succès = temps d'accès + temps pour déterminer si succès ou échec.



# Pourquoi ça marche ? Localité

- Principe de localité : à un instant donné, un programme accède à une portion relativement petite de l'espace d'adressage.

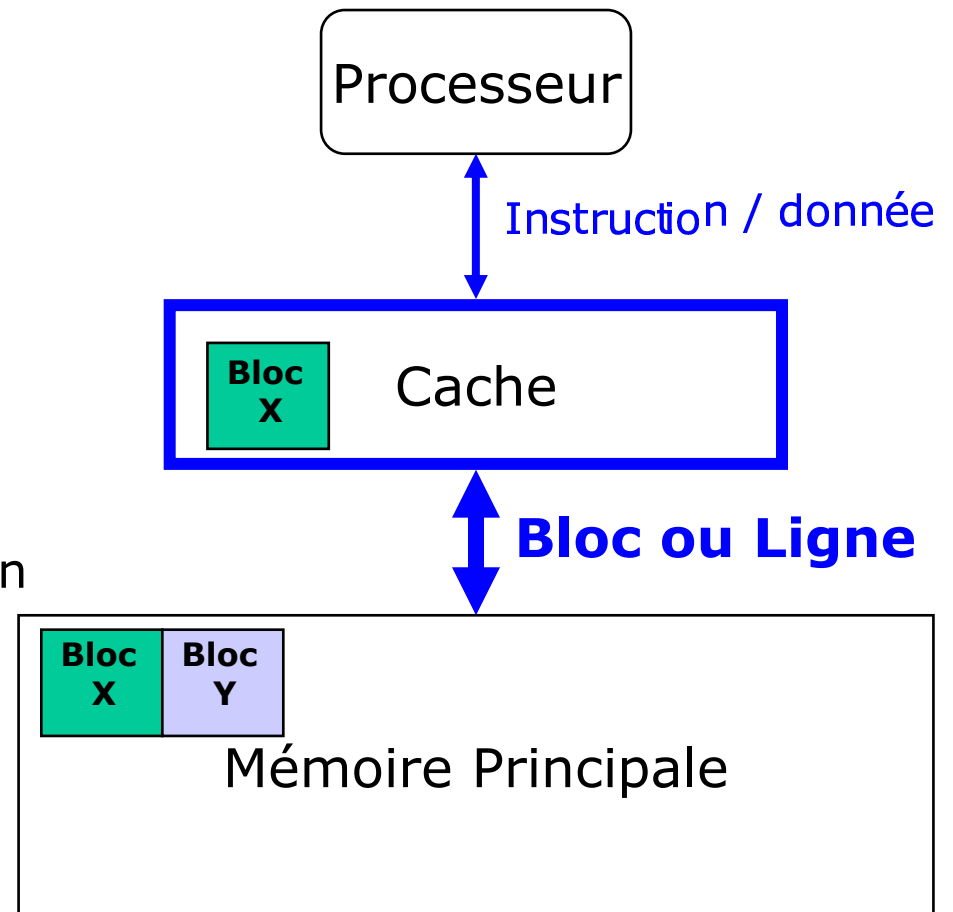


- **Localité temporelle** : adresse A référencée à T, alors forte probabilité de référencer A à T+t, t petit. Réutilisation.
- **Localité spatiale** : adresse A référencée, alors forte probabilité de référencer A+a, a petit.



# Exploitation de la localité

- Principe des caches : exploitation de la localité spatiale et de la localité temporelle.
- La localité temporelle est simplement exploitée en conservant une donnée dans le cache.
- La localité spatiale est exploitée en chargeant les données par **blocs** et non individuellement.



# Localités des données et des instructions

Les instructions, comme les données, possèdent de fortes propriétés de localité.

```
for (i=0; i<N; i++) {  
  for (j=0; j<N; j++) {  
    y[i] = y[i] + a[i][j] * x[j]  
  }  
}
```

- **y[i]**: propriétés de localités temporelle et spatiale.
- **a[i][j]**: propriété de localité spatiale.
- **x[j]**: propriété de localité temporelle et spatiale.

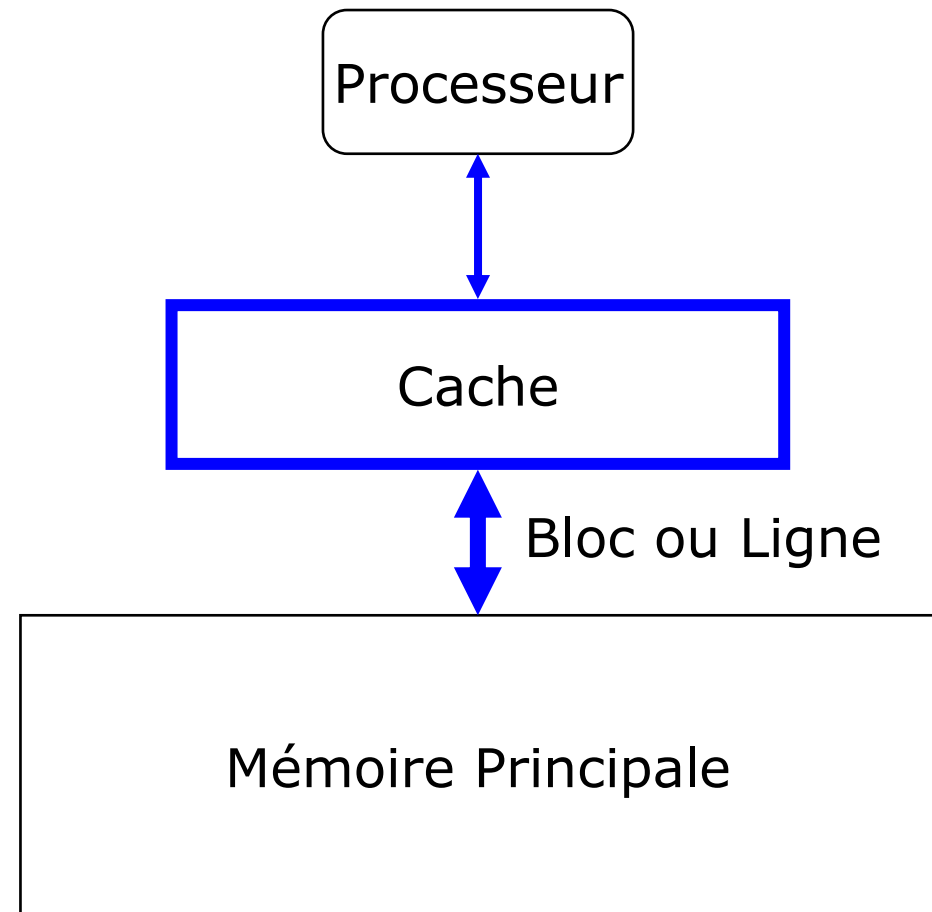
```
...  
05 LOOP      LDR R1, R0, #3  
06           ADD R1, R1, #5  
07           STR R1, R0, #30  
08           ADD R0, R0, #1  
09           ADD R3, R0, R2  
0A           BRn LOOP  
...
```

Boucle : réutilisation des instructions :  
localité temporelle

Instructions consécutives en mémoire :  
localité spatiale

# Caractéristiques d'un cache

- Taille de la ligne = taille du bloc.
- Taille du cache.
- Organisation du cache (associativité, politique de placement).
- Politique d'écriture.

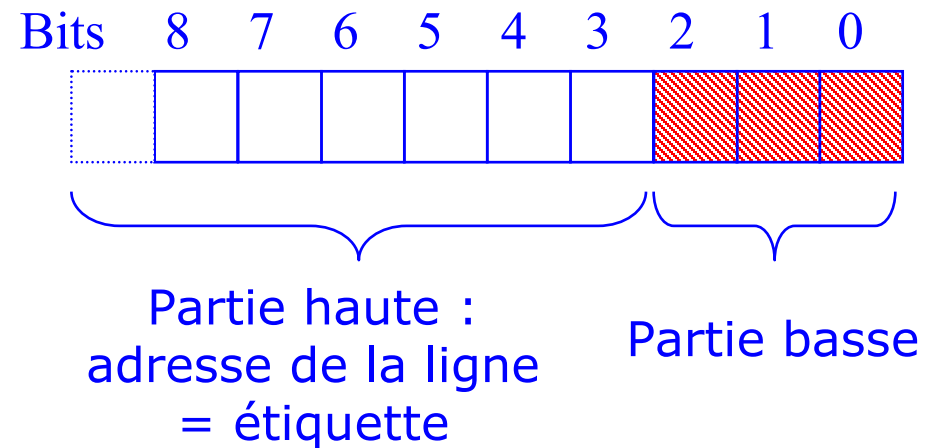


# Bloc ou ligne de cache

- Taille des transferts entre la mémoire et le cache.
- Décrire un bloc de données avec une seule adresse :
  - la partie haute de l'adresse des données d'une même ligne est identique,
  - seule la partie basse de l'adresse varie.
- Rappel: une adresse = 1 octet.

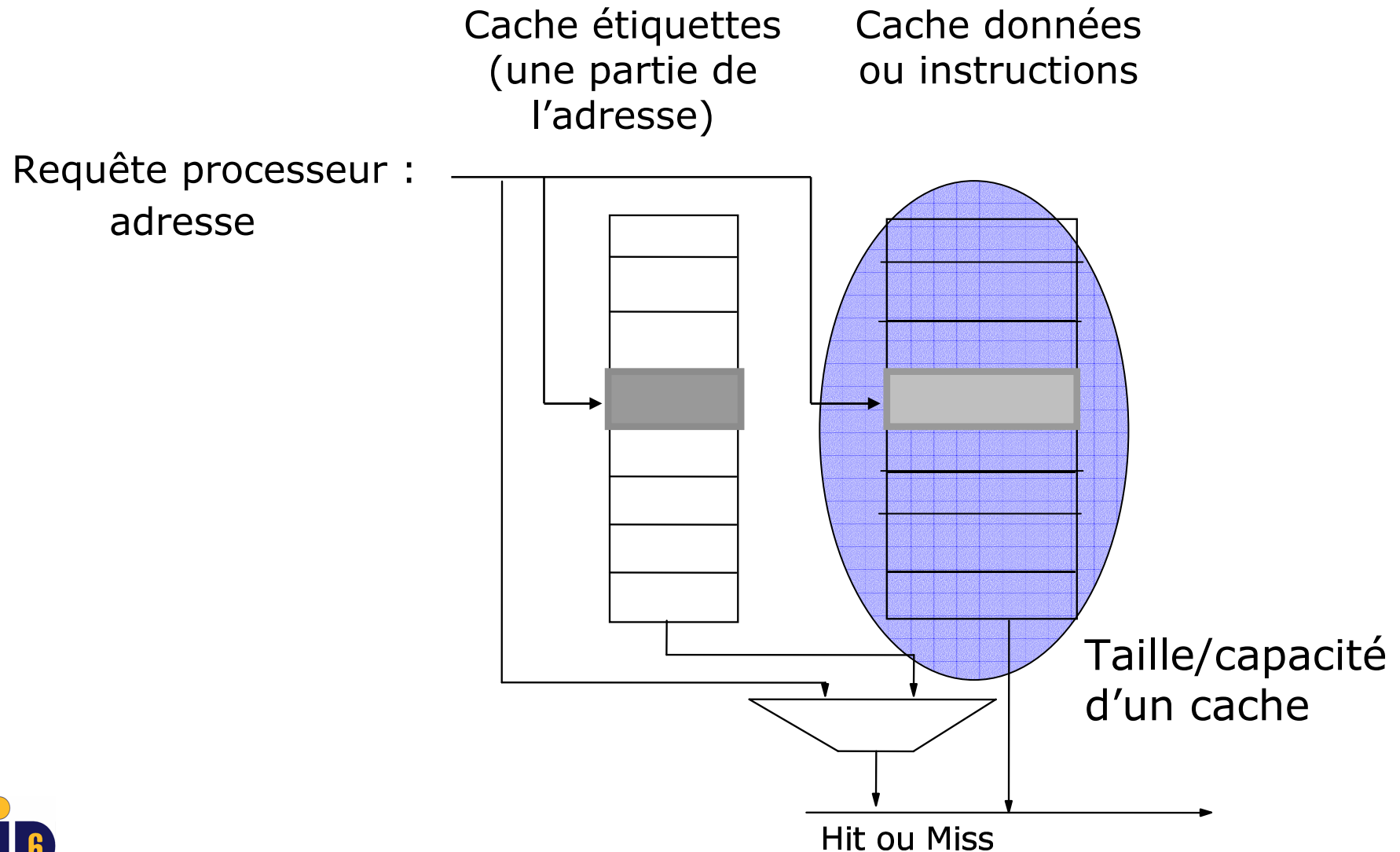
Exemple :

- Adresse sur 16 bits
- Bloc de 8 octets



0...010100000 }  
0...010100111 } Même ligne de cache  
0...010101000 } Lignes distinctes,  
adresses consécutives

# Structure générale d'un cache

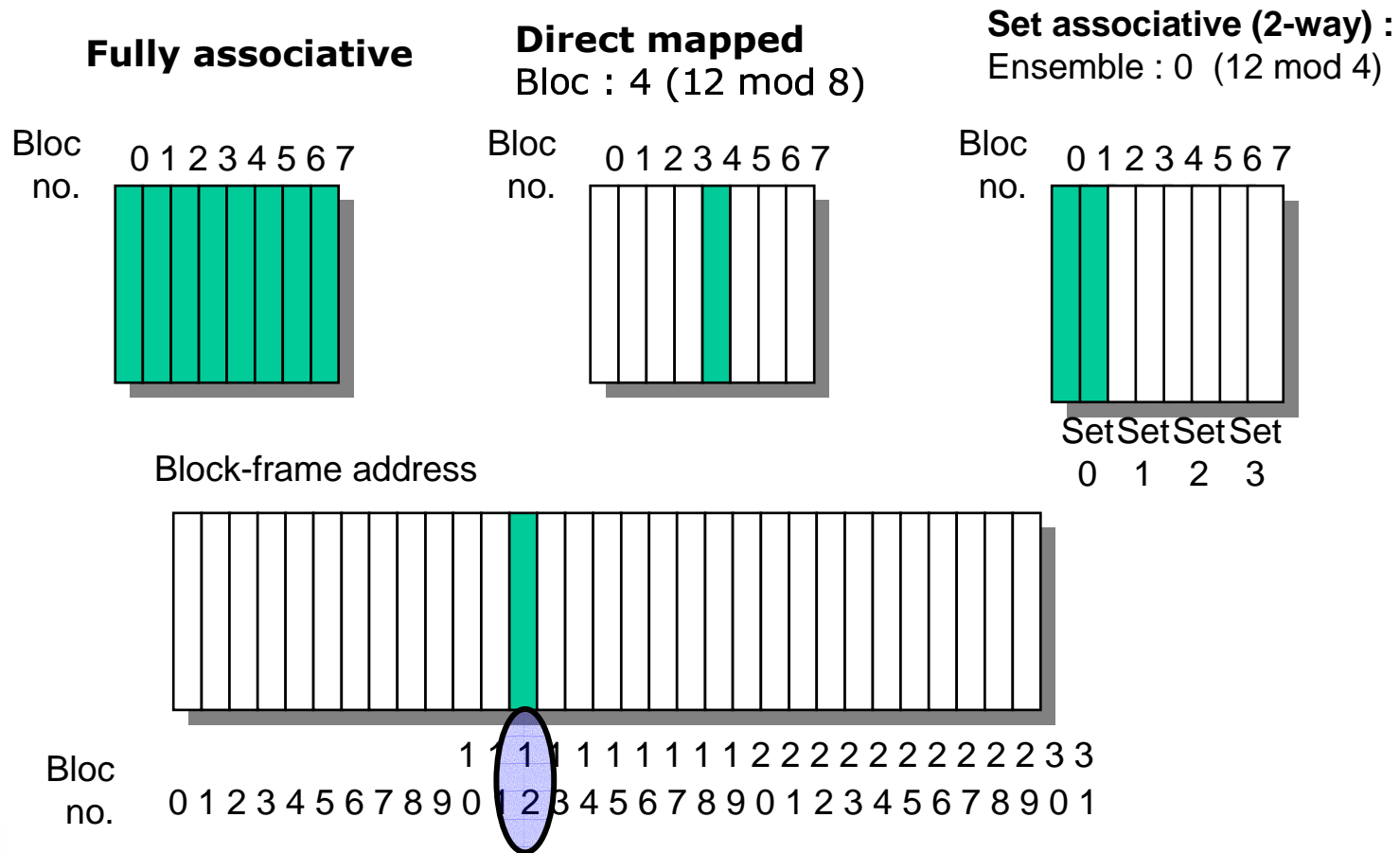


# Placement des données dans le cache

- Le placement des données dans le cache est géré par le matériel :
  - le programmeur n'a pas à se soucier du placement des données (contrairement à mémoire locale),
  - le fonctionnement du cache est transparent pour le programmeur.

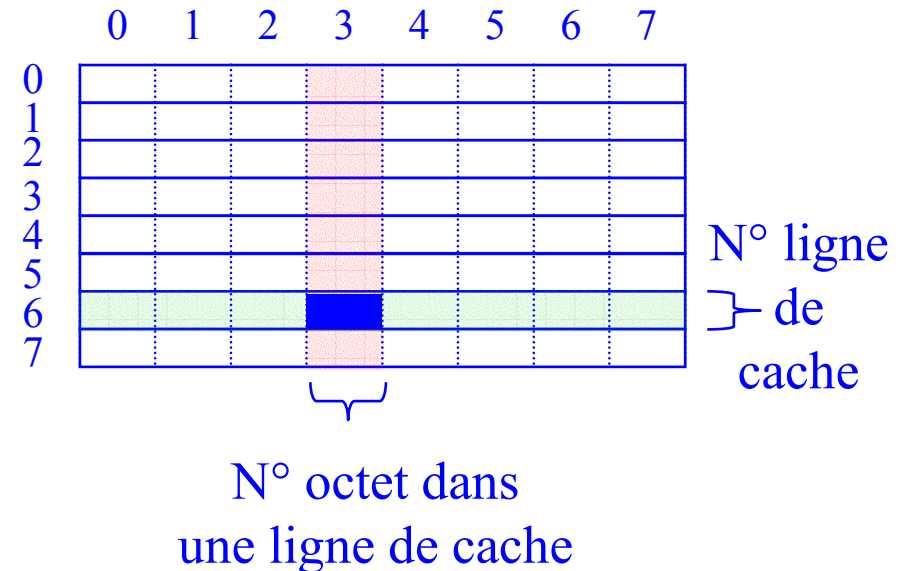
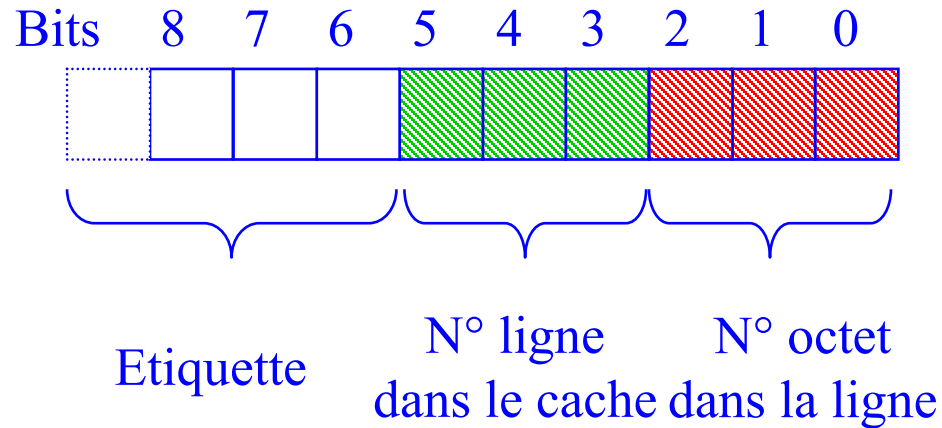
# Différentes organisations de cache

Où placer une ligne de la mémoire principale dans le cache ?



# Placement des données – Cache à correspondance directe

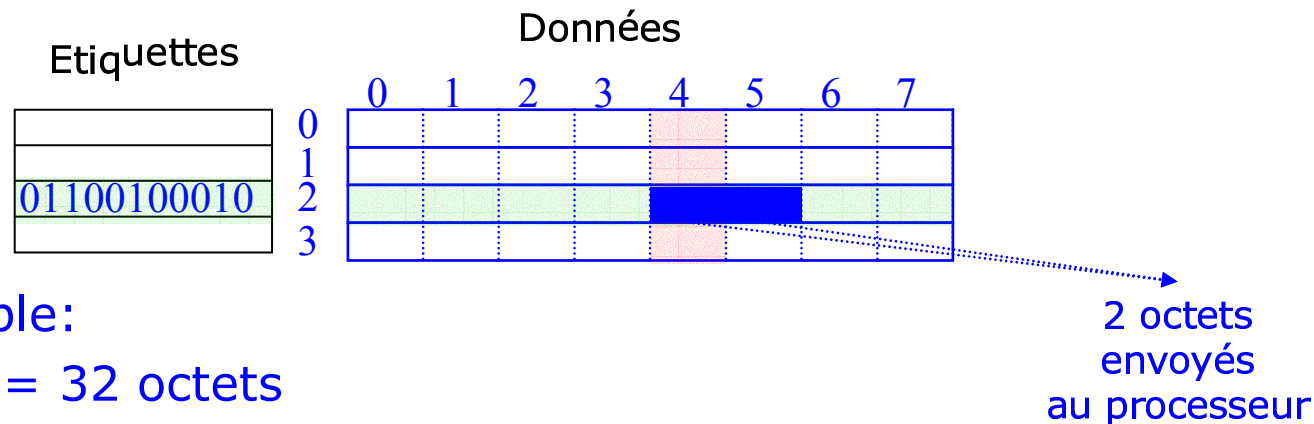
- L'emplacement d'une donnée est déterminé selon une fonction simple de l'adresse.
  - N° ligne dans le cache
  - N° d'octet dans la ligne



- Cache de  $C_S$  octets
- Ligne de  $L_S$  octets
- N° octet:  $\log_2(L_S)$  bits de poids faible de l'adresse.
- N° ligne:  $\log_2(C_S/L_S)$  bits de poids faible suivants de l'adresse.



# Lecture d'une donnée – Cache à correspondance directe



- Exemple:
  - $C_s = 32$  octets
  - $L_s = 8$  octets
- Adresse demandée (16 bits):
  - 01100100010**10100**
  - N° ligne: **10**
  - N° octet dans la ligne: **100**
- Une requête peut avoir une taille variable:
  - octet, demi-mot, mot...
  - requête = adresse + nombre d'octets
  - adresse = adresse du premier octet
  - exemple: 2 octets (mot de 16 bits)

# Intérêt de l'associativité

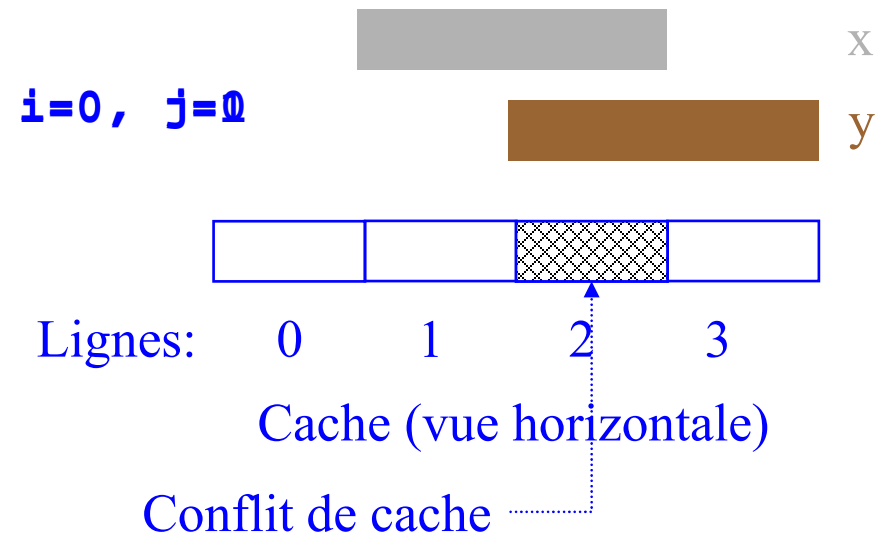
- Taille mémoire physique  $\gg$  taille cache.
- La fonction de placement peut engendrer des conflits entre les données.
- CS = 32 o, Ls = 8 o, x et y flottants double précision (8 o).

```
for (i=0; i<N; i++) {  
    for (j=0; j<2; j++) {  
        x[j] = y[j]  
    }  
}
```

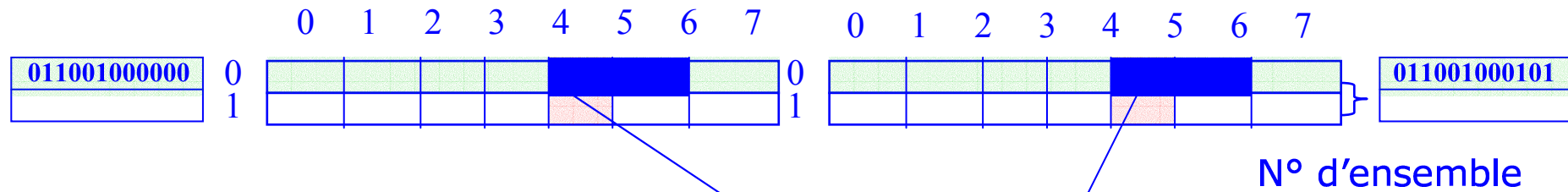
@x = 0110010001001000

@y = 0000100000001000

On peut réduire les conflits en augmentant l'**associativité** des caches.



# Lecture d'une donnée – Cache associatif

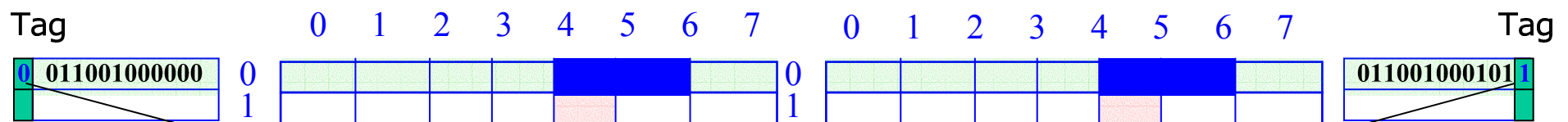


- Exemple :
  - $C_s = 32$  octets
  - $L_s = 8$  octets
  - $A = 2$
  - Requête = 2 octets
- Adresse demandée (16 bits) :
  - 011001000101**0100**
  - N° d'ensemble : **0**
  - N° octet dans la ligne: **100**

2 emplacements  
possibles

# Placement d'une donnée - Cache associatif

- Une donnée peut être stockée dans  $n$  emplacements différents.
- Il faut choisir la ligne dans laquelle on va charger la nouvelle donnée.
- Le choix est effectué entre les lignes du cache qui peuvent accueillir la donnée (l'ensemble):
  - LRU (*Least Recently Used*) : on choisit le bloc le plus anciennement accédé.
  - FIFO (*First In First Out*).
  - *Random*.
  - Pseudo-LRU : la ligne la plus récemment accédée n'est pas remplacée ; choix aléatoire entre les autres lignes.

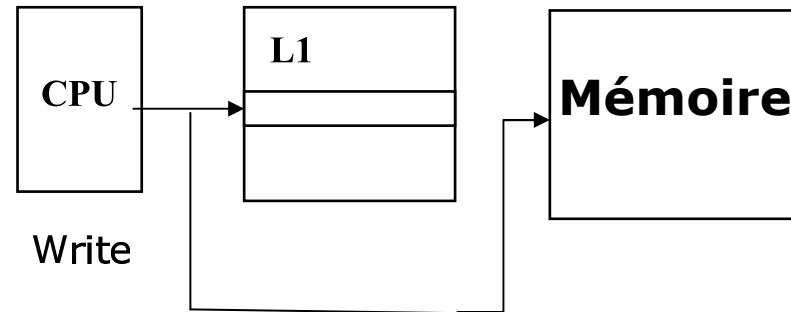


On choisit la + ancienne  
(exemple : Tag = 0) 20/83

# Politiques d'écriture en cas de succès

## Write Through

(cohérence au plus tôt)



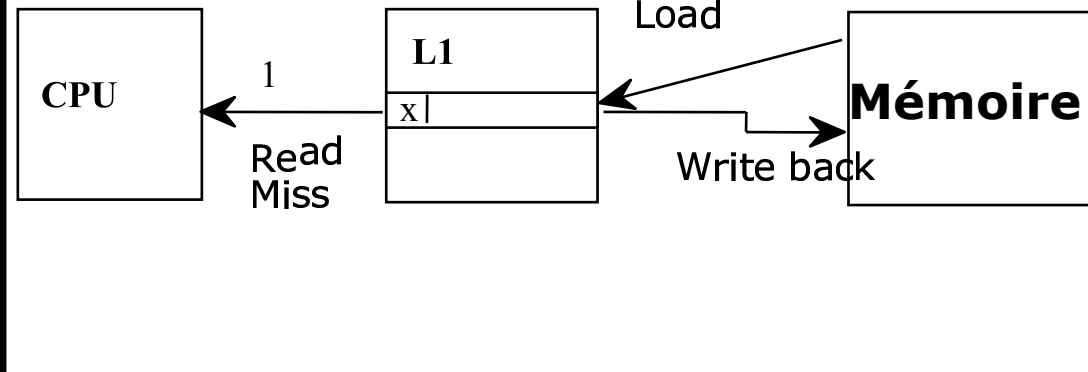
## Write Back

(cohérence au plus tard)

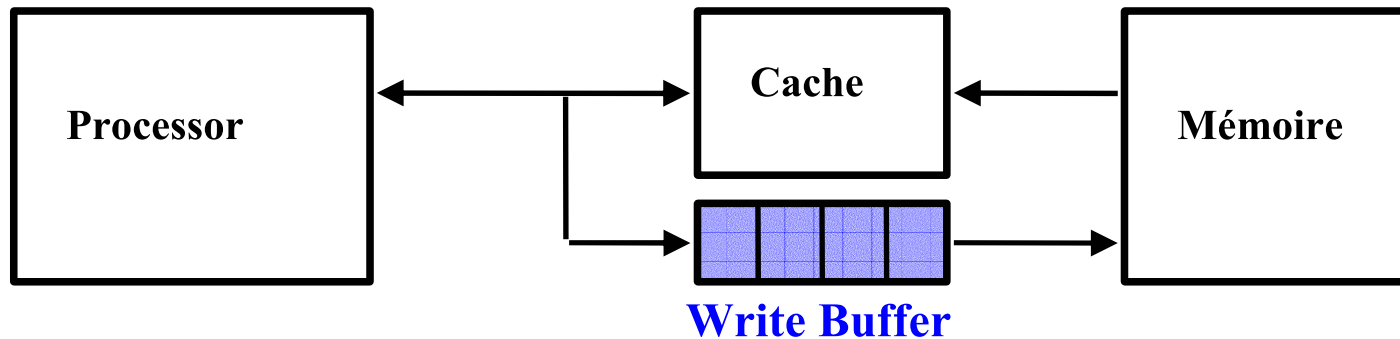


## Write back

(ligne *dirty* remplacée)

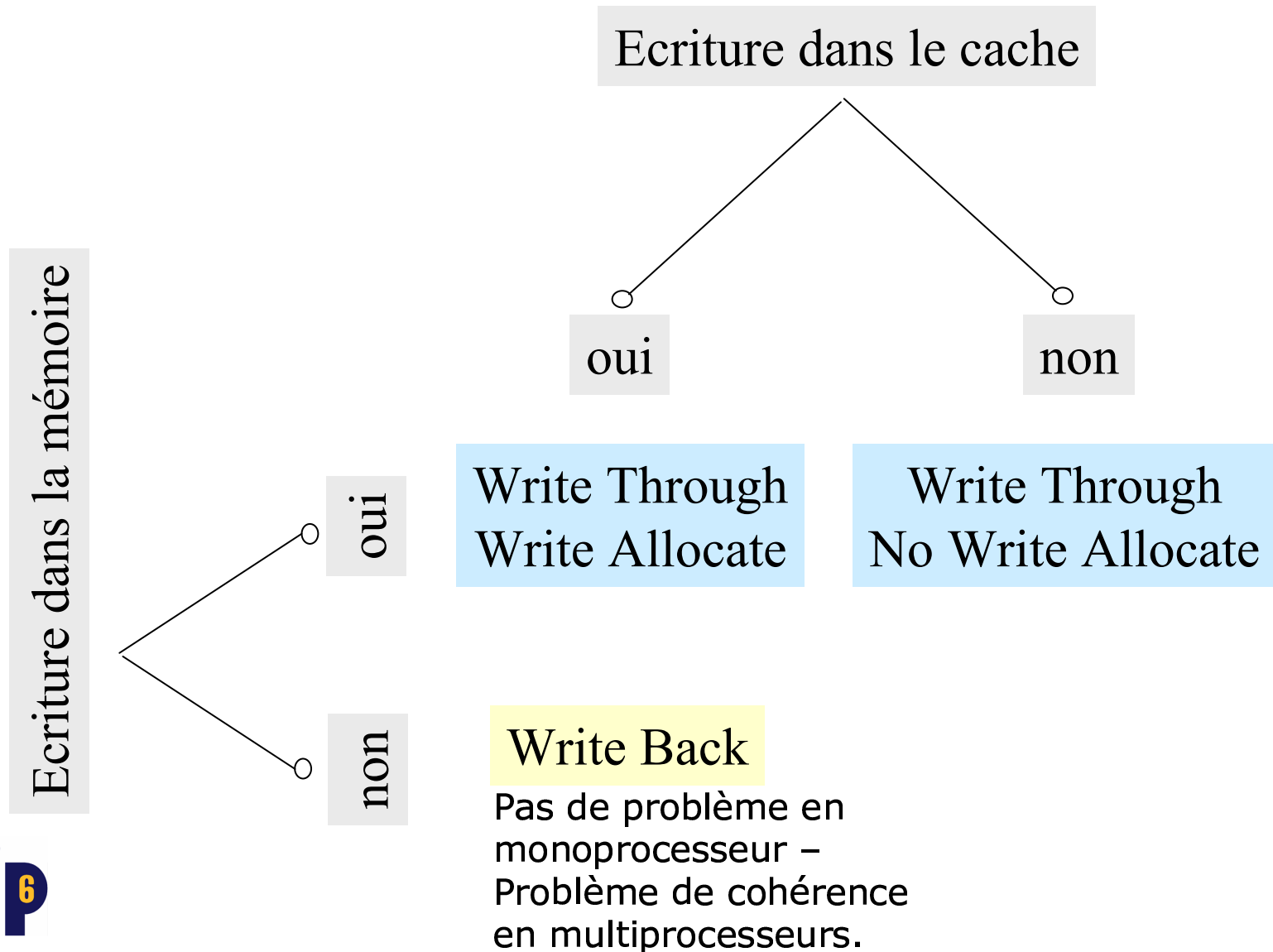


# Write Buffer pour la politique d'écriture Write Through



- *Write buffer* : FIFO entre le cache et la mémoire.
- Processeur : écrit les données dans le cache et dans le *write buffer*.
- Contrôleur mémoire : écrit le contenu du *write buffer* dans la mémoire.
- Avantage :
  - Regrouper les écritures en mémoire (burst).
  - Priorité lecture / écriture.

# Politique d'écriture en cas d'échec



# Cache performance

$$CPI_{\text{Cache}} = C \times \left( CPI_{\text{Cache}} + \frac{\text{Misses}}{h} \times M \times \text{Miss} \right) \times \text{CycleTime}$$

$$CPI_{\text{Cache}} = C \times \left( CPI_{\text{Cache}} + \frac{\text{Misses}}{h} \times \text{Miss} \right) \times \text{CycleTime}$$

$CPI_{\text{Execution}}$  inclus instructions ALU et mémoire.

## Performance de la hiérarchie mémoire AMAT = Average Memory Access Time

$$CPI_{\text{Cache}} = C \times \left( \frac{IP}{h} \times CPI_{IP} + \frac{\text{Misses}}{h} \times M \right) \times \text{CycleTime}$$

$$M = \text{Time} + M \times \text{Miss} \times y$$

$$= \left( \text{Time}_h + M_h \times \text{Miss}_h \times y_h \right) +$$

$$\left( \text{Time}_D + M_D \times \text{Miss}_D \times y_D \right)$$

$CPI_{\text{ALUOps}}$  n'inclus pas les instructions mémoire.



# Impact sur la performance

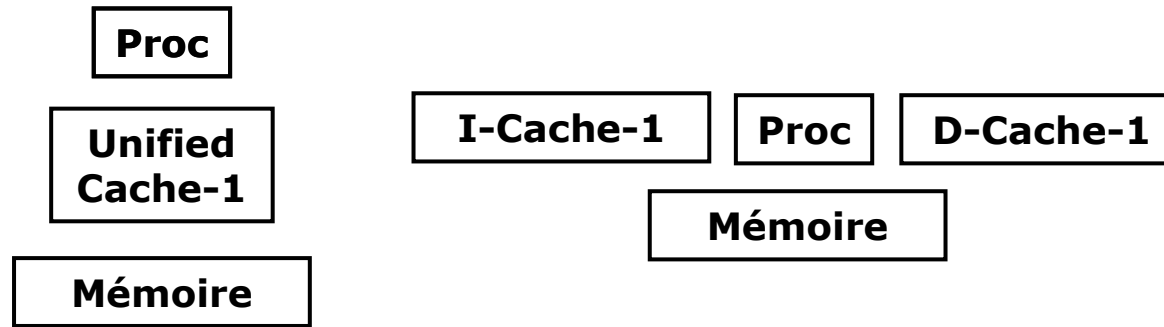
- Hypothèses :
  - Temps de cycle = 5 ns (fréquence = 200 MHz) - Idéal (sans échecs) CPI = 1.1 - Hit time = 1
  - 50% arith/logic, 30% Id/st, 20% contrôle
  - 10% des opérations mémoire (Id/st) - 50 cycles miss penalty
  - 1% des instructions - même miss penalty
- CPI = idéal CPI + average stalls per instruction
  - = 1.1 + [ 0.30 x 0.10 x 50 ] + [ 1 x 0.01 x 50 ]
  - = (1.1 + 1.5 + .5) = 3.1
- 64% du temps, le processeur attend la mémoire !
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2.54$

$$= \left( \frac{HitTime}{CPI} + \frac{MissRate}{CPI} \times MissPenalty \right) + \left( \frac{HitTime}{CPI} + \frac{MissRate}{CPI} \times MissPenalty \right)$$

# Impact sur la performance (suite)

- Hypothèses :
  - Idem slide précédant.
  - Variation du nombre d'échecs données : 100%, 50%, 10%, 1%, 0%.
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 1 \times 50] = 12,9$
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.5 \times 50] = 7,15$
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2,54$
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.01 \times 50] = 1,5$
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0 \times 50] = 1,38$

# Cache unifié versus caches séparés



- Exemple :
  - 16KB I&D: Taux de miss inst. = 0.64%, Taux de miss données = 6.47%
  - 32KB unifié : Taux de miss = 1.99%
- Comparaison :
  - 33% data ops.
  - Temps d'accès = 1, Pénalité d'échecs = 50.
  - Un succès data sur le cache unifié = 1 stall (un seul port de lecture).

$$AMAT_{\text{Harvard}} = (1/1.3) \times (1 + 0.64\% \times 50) + (0.3/1.3) \times (1 + 6.47\% \times 50) = 2.05$$

$$AMAT_{\text{Unifié}} = (1/1.3) \times (1 + 1.99\% \times 50) + (0.3/1.3) \times (1 + 1 + 1.99\% \times 50) = 2.24$$

# Optimisation des mémoires cache

# Améliorer les performances des caches

- Temps d'accès moyen à la mémoire :  
$$\text{AMAT} = \text{Temps de succès} + \frac{\text{Taux d'échecs} * \text{Pénalité d'échecs}}{\text{Taux d'échecs}}$$
- 3 moyens d'améliorer la performance :
  - **Réduire le taux d'échecs,**
  - **Réduire la pénalité d'échecs,**
  - **Réduire le temps d'accès au cache** (cas d'un succès).
- Un autre moyen lié à l'exploitation de l'ILP :
  - **Augmenter le débit d'instructions.**

# Quelques techniques pour réduire le taux d'échecs

- Augmenter la taille du cache.
- Augmenter la taille des  $do^{CS}$ .
- Augmenter l'associativité.
- Le *Victim Cache*.
- Les caches Pseudo-Associatifs.
- Les caches *skew associative*.
- Le préchargement matériel (instructions et données).
- Des optimisations à la compilation.
- ...

# Les types d'échecs : les 3Cs

## (un peu de vocabulaire)

- **Compulsory misses** (cold/startup misses). Le premier accès à un bloc absent du cache. Echec même en cas de cache infini.
- **Capacity misses** (ensemble de données trop grand). Echecs sur une même donnée. Echec même en cas de cache complètement associatif.
- **Conflict ou interference misses** (degré d'associativité insuffisant). Echec du à l'utilisation d'un même ensemble (bloc).
- **Un 4ème C : Coherence**. Echec causé par la gestion de la cohérence de cache.

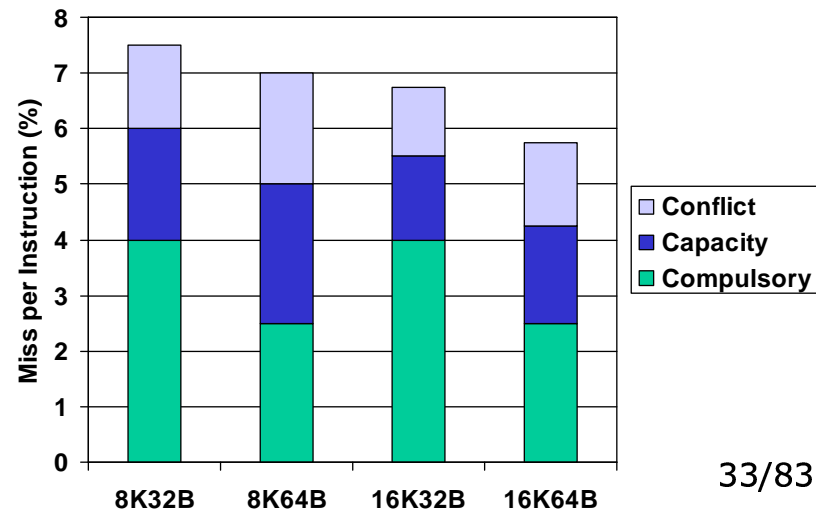
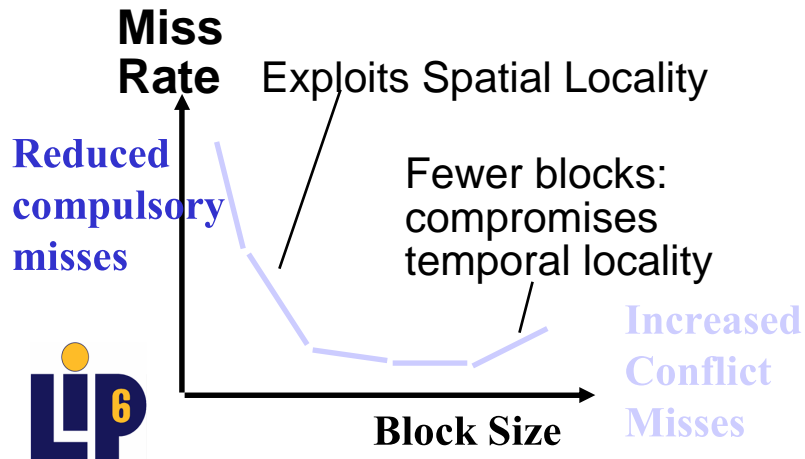
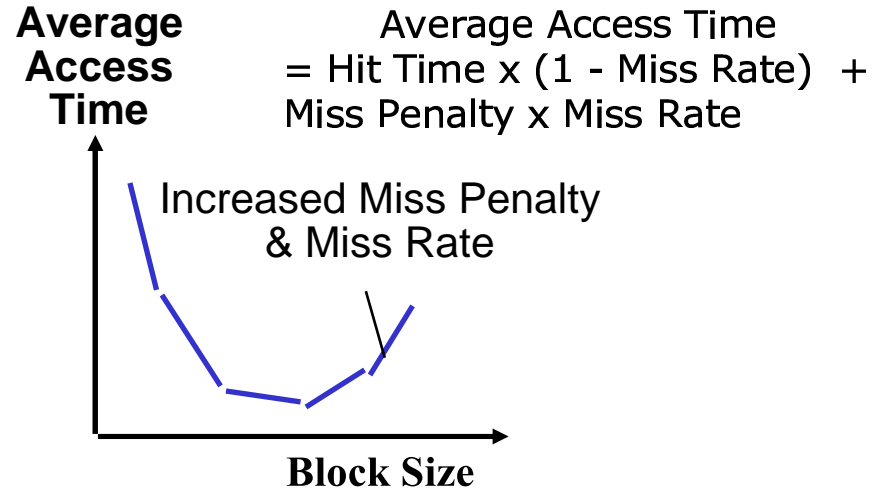
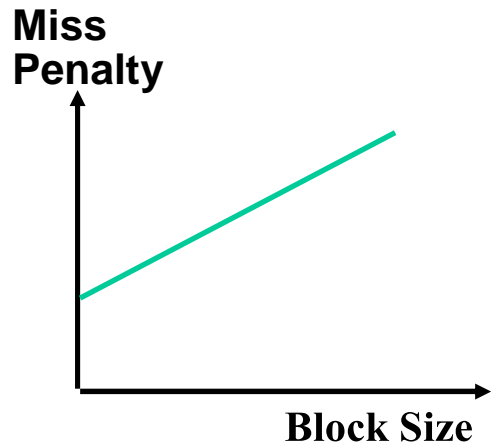
# Augmenter la taille du cache

- Pas d'effet sur les *compulsory misses* (échec 1er accès).
- Réduit les *capacity misses* (plus de capacité !).
- Réduit les *conflict misses* (en général) :
  - Peu de blocs en concurrence en moyenne pour un ensemble (beaucoup d'ensembles).
  - Probabilité plus faible que le nombre de blocs actifs d'un ensemble  $>$  taille de l'ensemble.
- Mais augmentation du temps d'accès (et du coût).



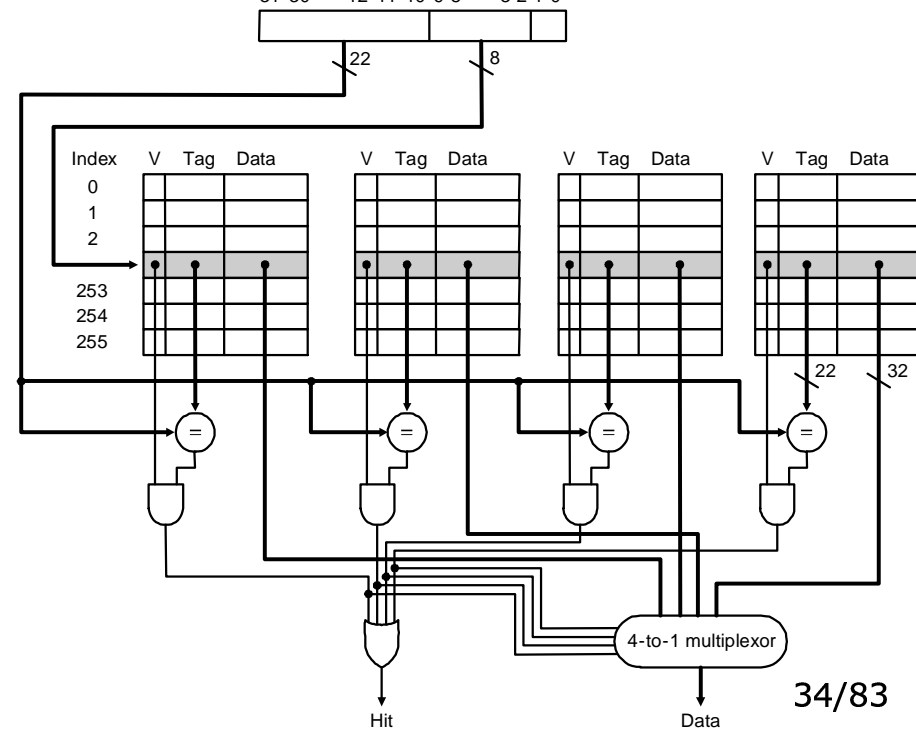
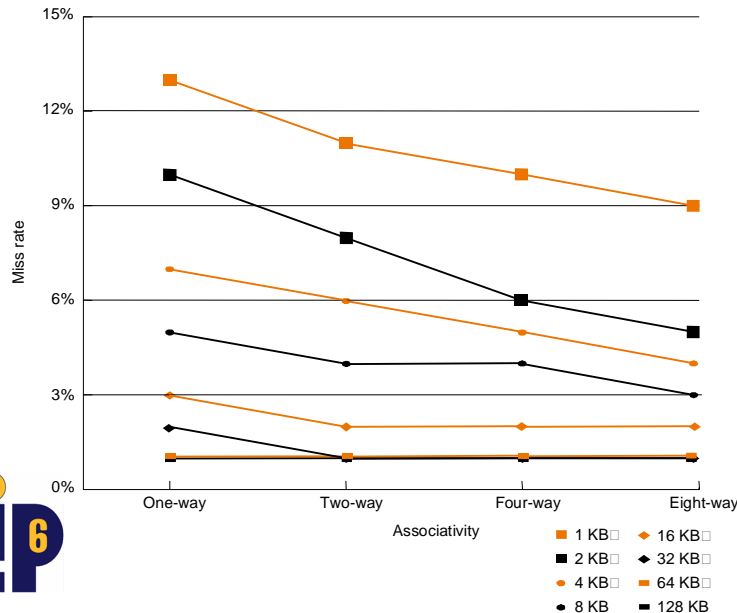
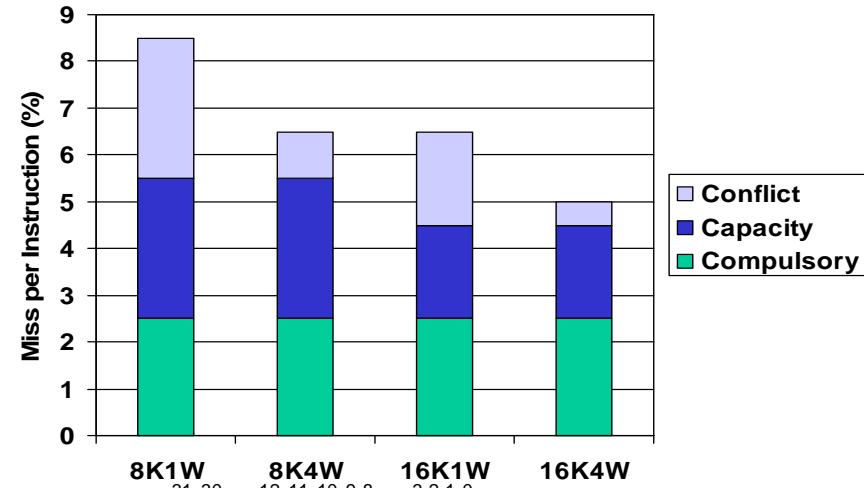
# Augmenter la taille du bloc/ligne

- En général, un bloc large exploite la localité spatiale (réduit compulsory misses), mais :
  - Augmentation du temps de transfert.
  - Moins de blocs pour une taille de cache donnée, augmentation des *conflict misses*.



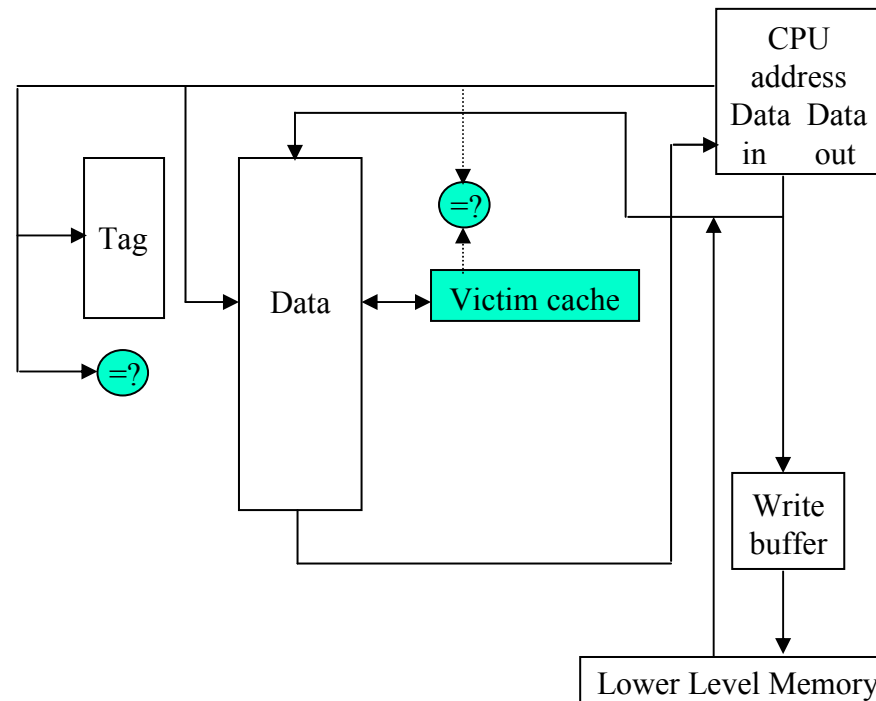
# Augmenter l'associativité

- Avantages :
  - Réduction des *conflict misses*.
- Désavantages :
  - Augmentation du temps d'accès,
  - Nécessite plus de matériels (comparators, muxes, tags),
  - Diminution des bénéfices - passage 4- ou 8- way.



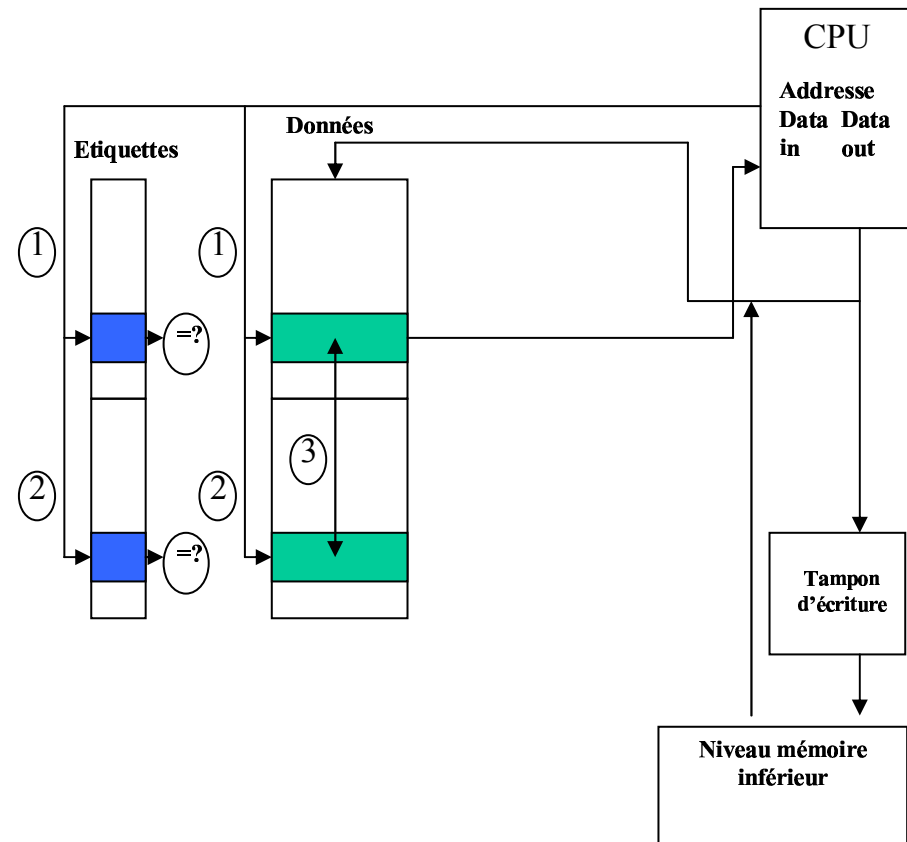
# Le Victim Cache

- Réduction des *conflicts misses*.
- Ajout d'un buffer permettant de stocker les données évincées du cache.
- Jouppi [1990]: "4-entry victim cache removed 20% to 95% of conflicts for a 4-KB direct-mapped data cache".
- *Victim cache* « original » dans HP PA7200 (petit cache, fully-associative).
- Ancêtre du cache L2 on-chip...



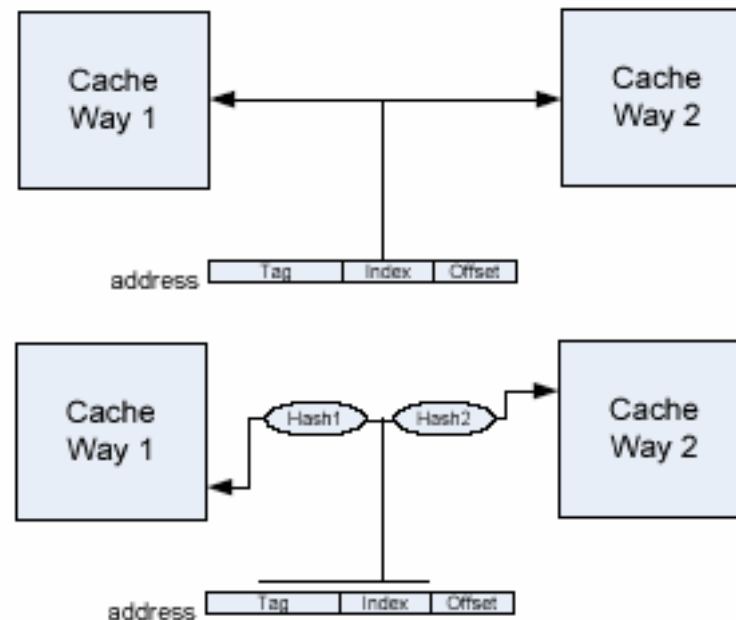
# Les caches pseudo-associatifs

- Réduction des *conflicts misses*.
- Nombre d'échecs similaires à un cache *2-way* et temps d'accès à un cache *direct-mapped*.
- En cas d'échec sur 1, accès à 2.
- + sophistiqué (*way prediction*) : stocke des bits de prédiction pour décider quelle partie du cache est accédée en premier.
- Utilisé dans le MIPS R1000 L2 cache.



# Les caches *skew associative*

- *Conflicts misses* quand  $n+1$  blocs (associativité =  $n$ ) ont le même index.
- Pour réduire ces *conflicts misses*, utiliser indices différents entre les voies du cache.
- Exemple : XOR index bit et tag bits.

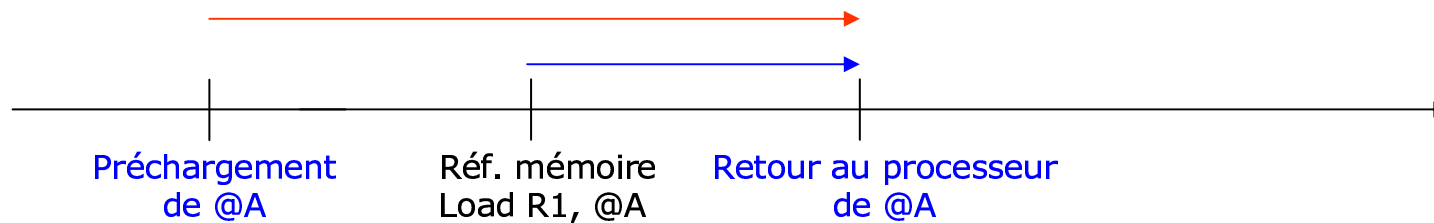
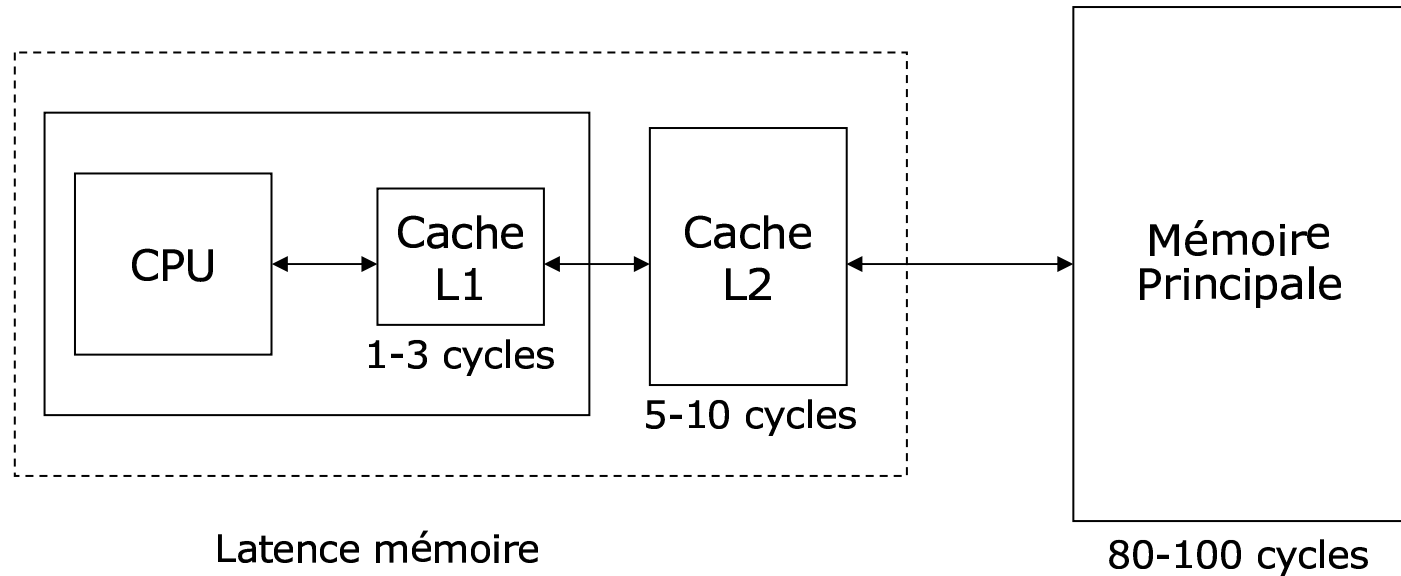


# Le préchargement matériel

- **Précharger.** Anticiper la demande explicite de la donnée par le processeur.

...

Add R1, R2, R3  
Or R3, R2, R4  
**Load R1, @A**  
Load R2, @B



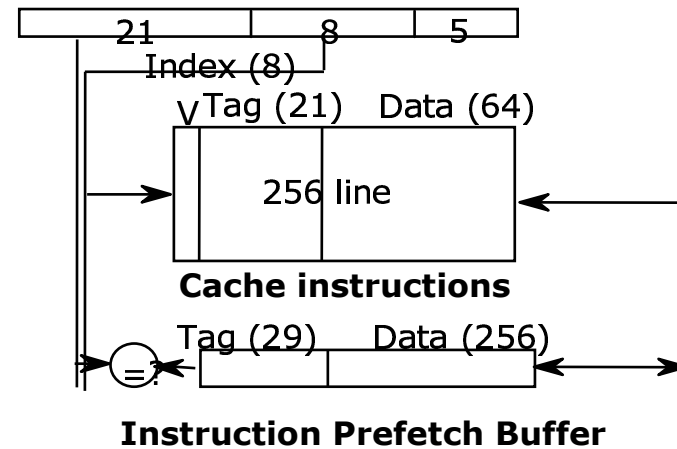
- **Prédire** quelles données le processeur va utiliser.

# Préchargement : prédire et stocker

- La technique de préchargement mise en oeuvre doit répondre à plusieurs questions :
  - **What** : quelles données préchargées (adresses),
  - **When** : quand précharger ces données,
  - **Where** : où stocker les données préchargées.

# Préchargement des instructions

- Lors d'un échec sur le cache instruction, la ligne requise est retournée au cache instructions et la ligne suivante est préchargée dans un tampon (exemple : 21064).



- Plus sophistiqué : la prédiction de branchement dirige le préchargement.

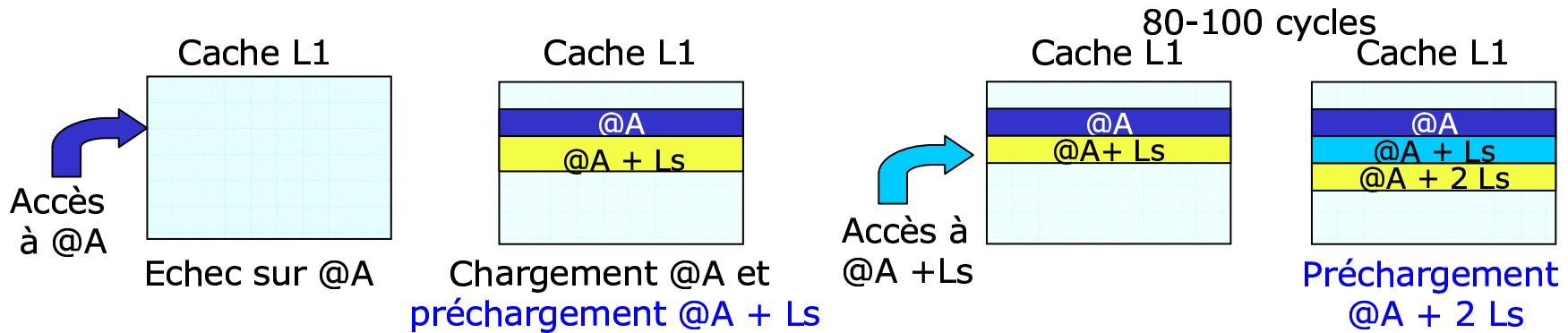
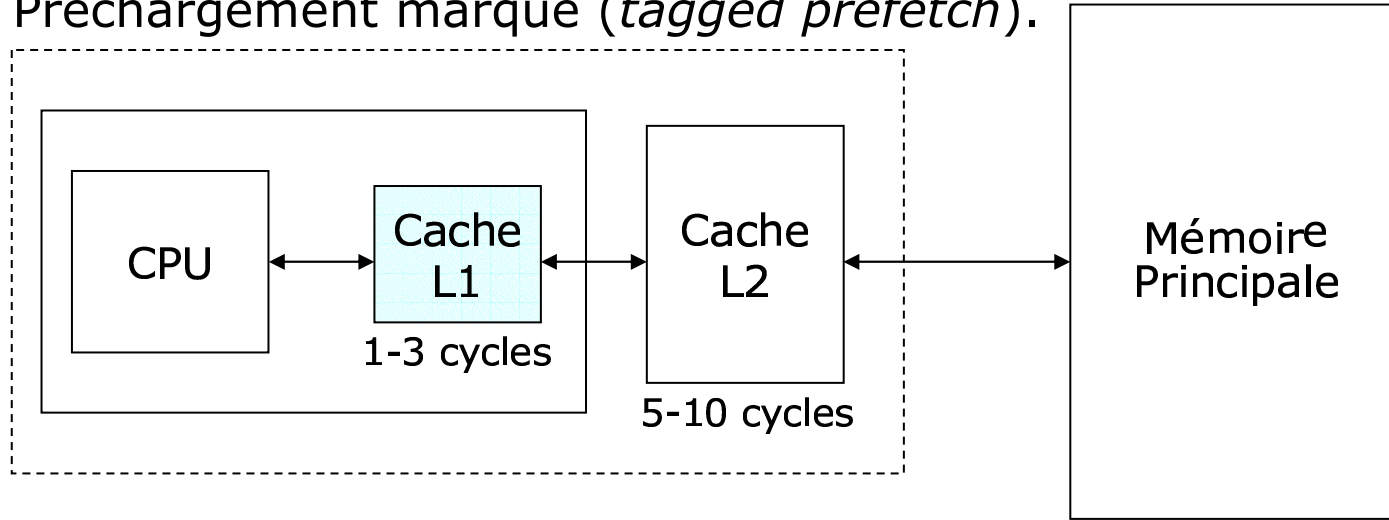


# Préchargement des données

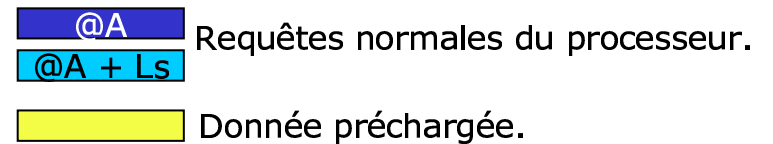
- Mécanismes de prédiction : sur succès, sur échec, préchargement marqué, table avec pas, ...
- Niveau de cache où est appliquée la prédiction : L1 ou L2.
- Où charger les données préchargées : tampon ou cache.

# Préchargement des données : un exemple de prédiction

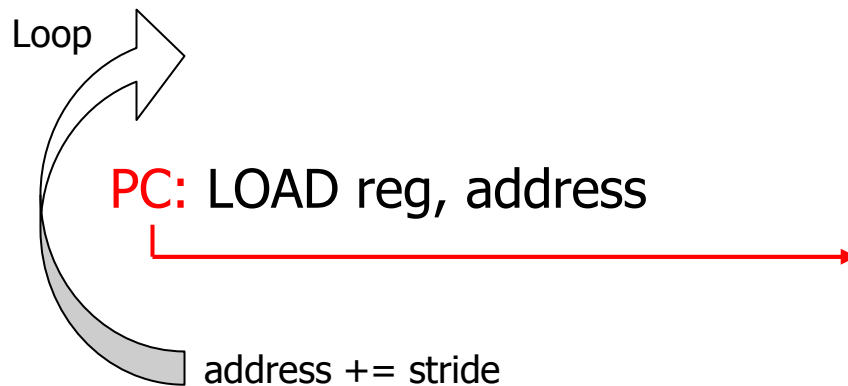
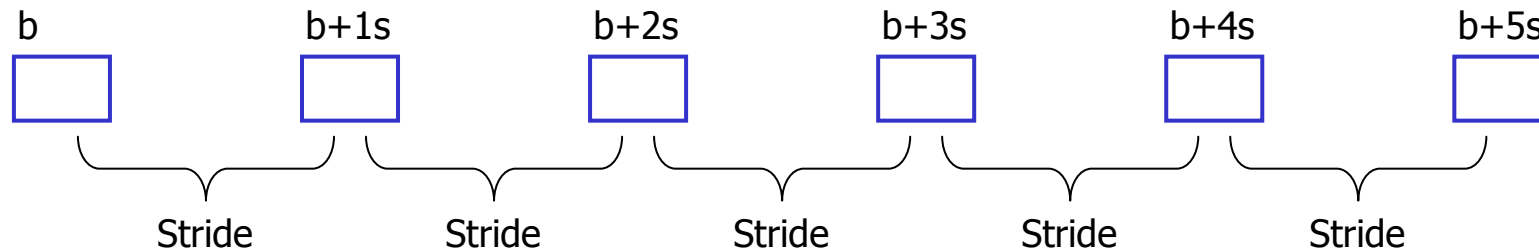
- Préchargement marqué (*tagged prefetch*).



- Préchargement sur échec + lors d'un accès à une donnée préchargée.



# Préchargement des données : un exemple de prédiction



Reference Prediction Table (RPT)

PC Tag	Prev. Address	Stride	State

# Des optimisations à la compilation

Objectif : augmenter la localité spatiale et temporelle des codes.

- Regroupement de tableaux (*merging arrays*),
- Permutation de boucles (*loop interchange*),
- Fusion de boucles (*loop fusion*),
- *Blocking*,
- *Padding*,
- Préchargement (charger plus tôt),
- ...

# Regroupement de tableaux

```
/* Avant : 2 tableaux séquentiels */
```

```
int val[SIZE];  
int key[SIZE];
```

```
/* Après : 1 tableau de structures */
```

```
struct merge {  
    int val;  
    int key;  
};  
struct merge merged_array[SIZE];
```

- Réduction des conflits entre val & key ; augmentation de la localité spatiale.

# Permutation de boucles

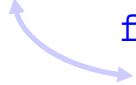
- Changer l'ordre des boucles afin d'accéder aux données dans l'ordre dans lequel elles sont stockées en mémoire.

**/\* Avant \*/**

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

**/\* Après \*/**

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



- Accès séquentiel versus accès avec pas de 100 ; augmentation de la localité spatiale.

# Fusion de boucles

- Combiner des boucles indépendantes (avec même compteur et des variables identiques).

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

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

- Potentiellement, 2 misses par accès à  $a$  &  $c$  versus 1 miss par accès ; augmentation de la localité temporelle.

# Multiplication de matrices

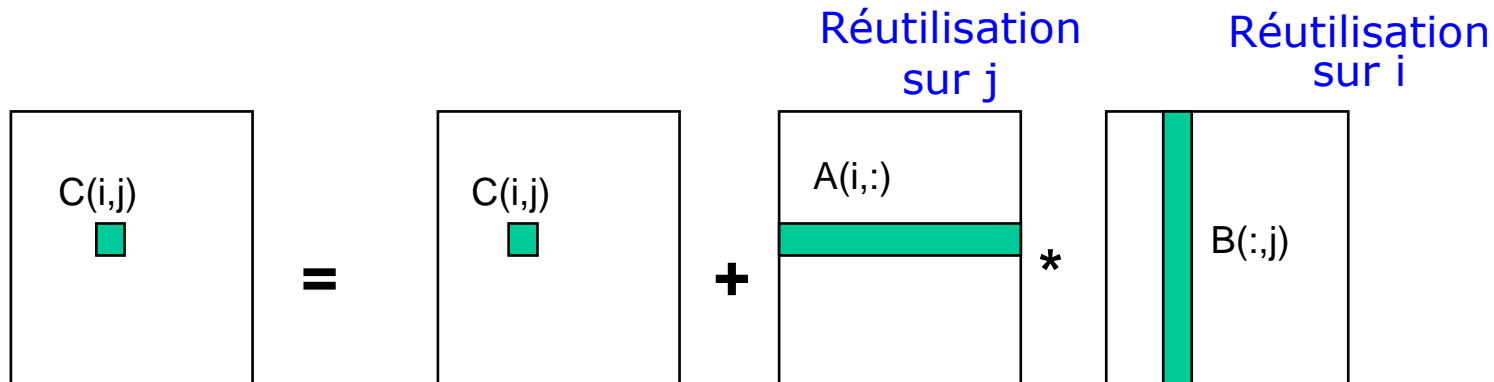
```
/* Calcul de C = C + A*B */
```

```
for i = 1 to n
```

```
  for j = 1 to n
```

```
    for k = 1 to n
```

```
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



- Si le cache ne peut pas contenir B entièrement, il y aura un miss à chaque accès.
- Si le cache ne peut pas contenir une ligne de A entièrement, il y aura un miss à chaque ligne.



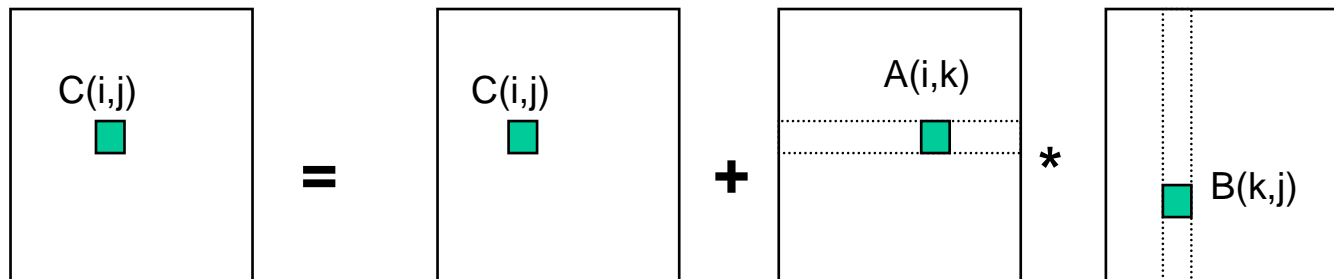
# Blocking

- Adapter la granularité à la hiérarchie mémoire.
- Calculer par blocs : on considère que les matrices A, B et C sont des matrices  $N \times N$  composées de sous-blocs  $b \times b$  où  $b = n / N$

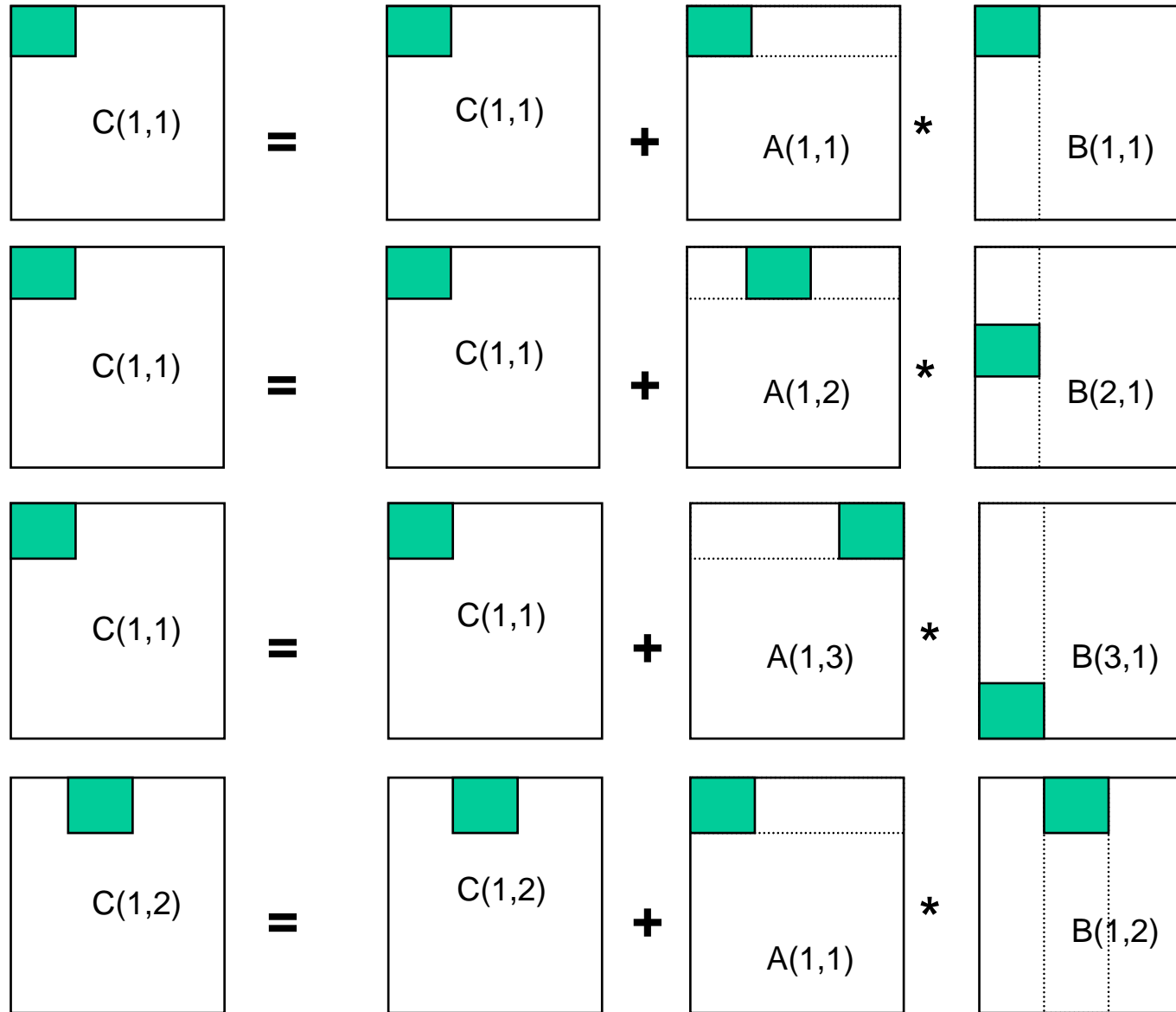
```

for i = 1 to N
  for j = 1 to N
    for k = 1 to N
      {read block A(i,k) into fast memory}
      {read block B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    {do a matrix multiply on blocks}
  {write block C(i,j) back to slow memory}

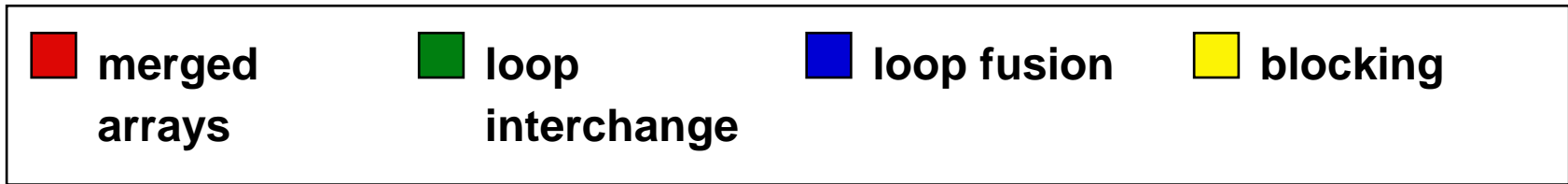
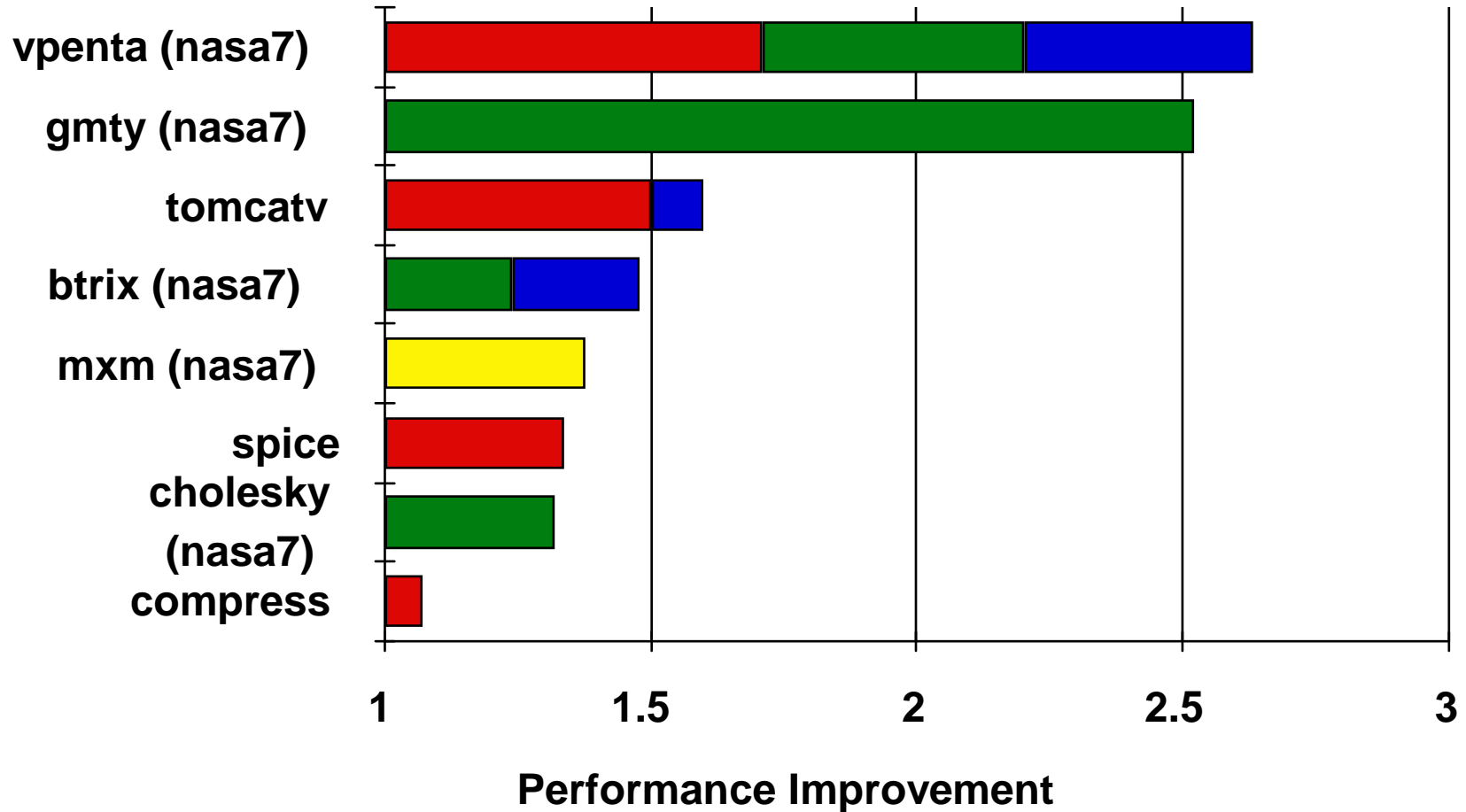
```



# Multiplication matrices bloquées



# Performance des optimisations



# Array padding

- Eviter les *conflict misses*.
- Taille du cache = 8192 octets, taille des réels = 8 octets, taille du bloc = 32 octets.

**/\* Avant \*/**

```
integer CS = 1024
real A(CS), B(CS)
for i = 1,CS do
  A(i) = A(i) +  $\beta$  * B(i)
```

**/\* Après \*/**

```
integer CS = 1024
real A(CS), pad(4), B(CS)
for i = 1,CS do
  A(i) = A(i) +  $\beta$  * B(i)
```

# Préchargement logiciel

- Les jeux d'instructions comportent des instructions spéciales permettant de lancer le préchargement. Instructions générées par le compilateur à la suite d'analyse de dépendances et de flot.

**/\*Avant \*/**

```
for i = 1,n do  
    A(i) = A(i) +  $\beta$  * B(i)
```

**/\*Après\*/**

```
Prefetch A(1), B(1), and  $\beta$   
for i = 1,n do  
    A(i) = A(i) +  $\beta$  * B(i)  
    Prefetch A(i+k), B(i+k)
```

# Quelques techniques pour réduire la pénalité d'échecs

- Un second niveau de cache
- Redémarrage précoce et mot demandé en premier
- Le *Victim Cache*
- Les caches non-bloquants
- Accélérer la translation d'adresses,
- ...

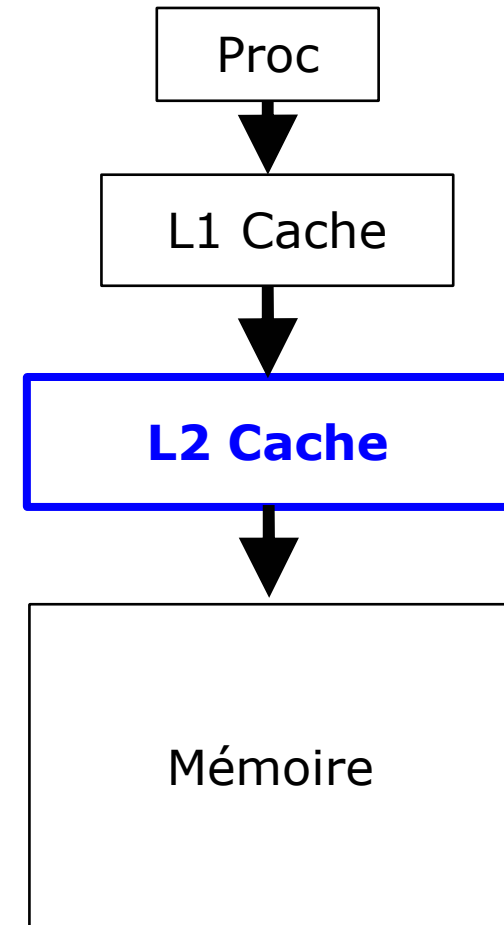
# Cache de second niveau

- Caches populaires – pénalité d'échecs = 10 cycles.
- Actuellement, pénalité d'échecs = 100 cycles.
- Comment masquer la latence mémoire ?  
Ajouter une mémoire cache entre le cache et la mémoire.
- Second niveau de cache plus grand - meilleur taux de succès. Temps de succès << pénalité d'échec.

$$AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

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

$$AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$



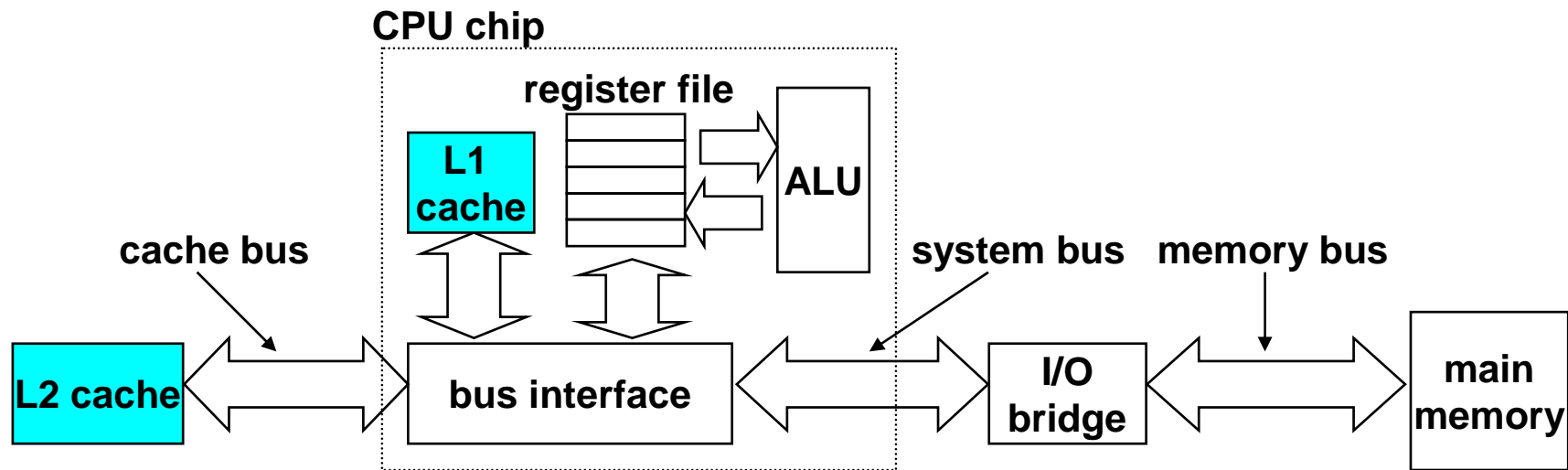
# Réduire la pénalité d'échecs Cohérence entre les niveaux de cache

Propriété d'inclusion entre les niveaux de cache :

- **Inclusif** : cache L1 est un sous-ensemble du cache L2.
  - Mécanisme de cohérence simplifié.
  - Taille de stockage effective = taille du cache L2.
  
- **Exclusif** : caches L1 et L2 exclusifs.
  - Taille de stockage effective = taille du cache L1 + taille du cache L2.
  - Forcer les invalidations (dans le cache L1 si invalidations L2).



# Hiérarchie mémoire typique avec interface



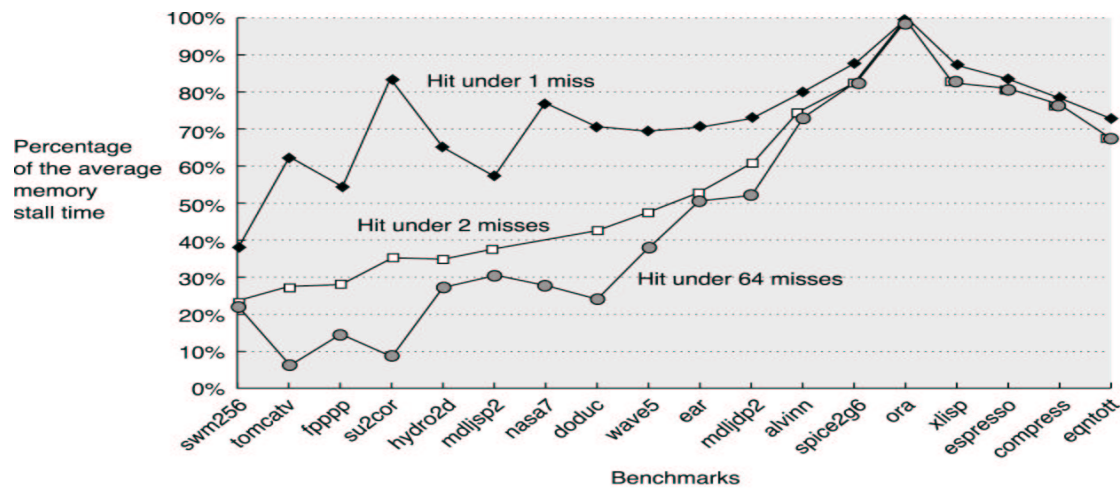
# Redémarrage précoce et mot demandé en premier

- Ne pas attendre que tout le bloc mémoire soit chargé pour utiliser les données :
  - **Redémarrage précoce** : dès que la donnée nécessaire est chargée, l'utiliser.
  - **Mot demandé en premier** : charger en premier la donnée ayant produit l'échec.
- En général, utile quand la taille des blocs est grande.

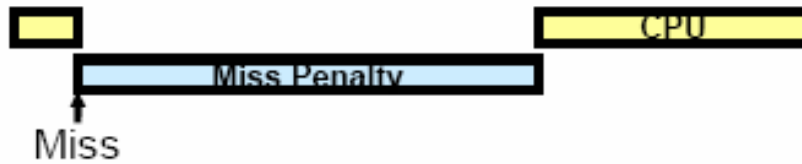
# Cache non-bloquant

- Cache non bloquant (non-blocking cache ou lockup-free cache).
- Lors d'un échec, le cache continue à traiter des instructions.
  - *Hit under miss.*
  - *Hit under multiple miss ou miss under miss.*
- Augmentation de la complexité du contrôleur de cache (maintenir plusieurs requêtes mémoire en suspend).

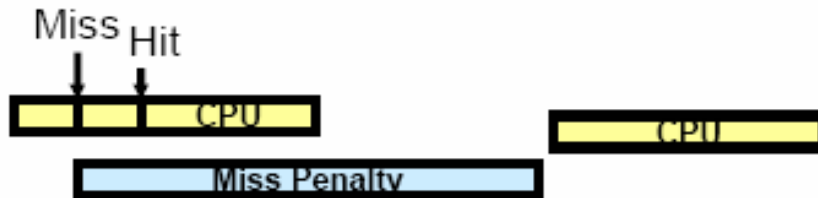
Ratio average stall time non blocking/ blocking cache



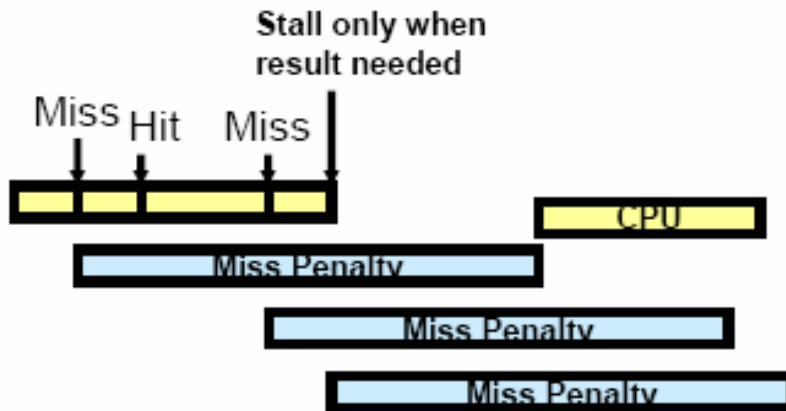
# Cache non-bloquant (exemples)



Stall sur miss



Hit sur un miss

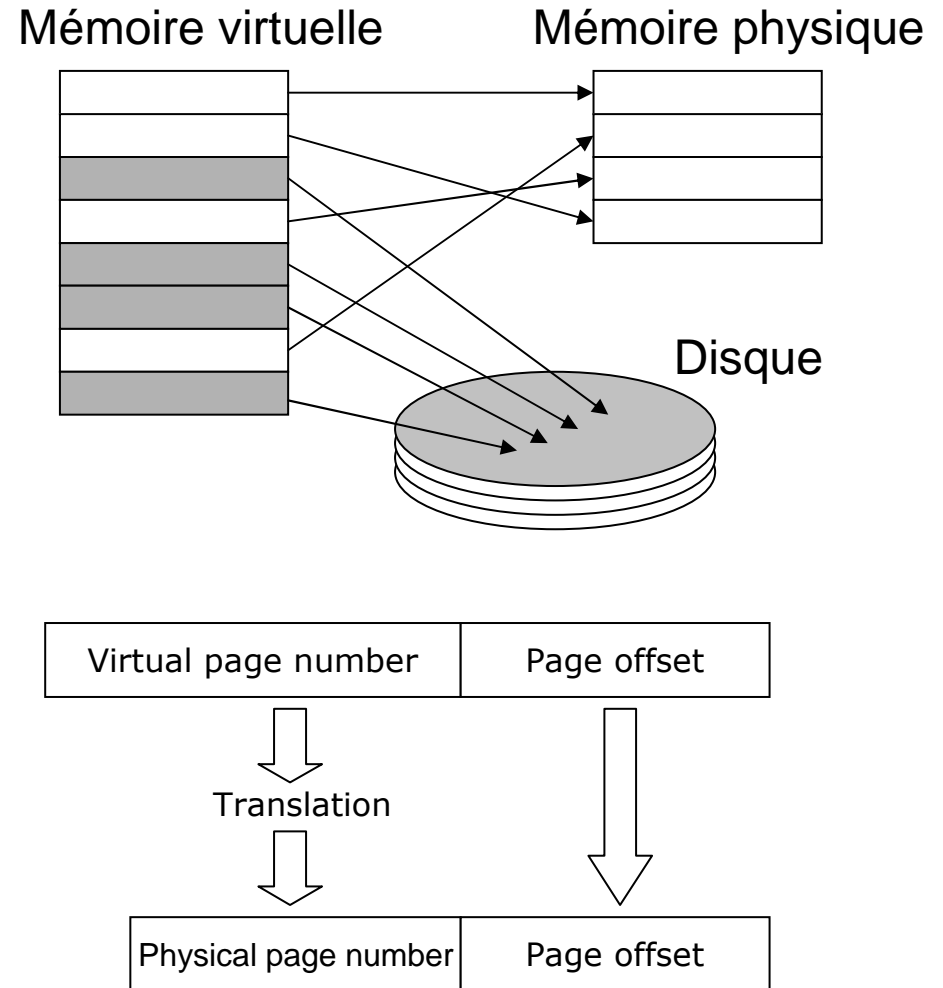


Plusieurs miss en suspend

# Accélérer la translation d'adresses

Réduire la pénalité d'échecs

- Mémoire adressable < mémoire physique -> mémoire virtuelle.
- MMU (*Memory Management Unit*) : gestion de l'espace mémoire et de la traduction de l'adresse virtuelle (espace adressable par le processus) vers l'adresse physique.

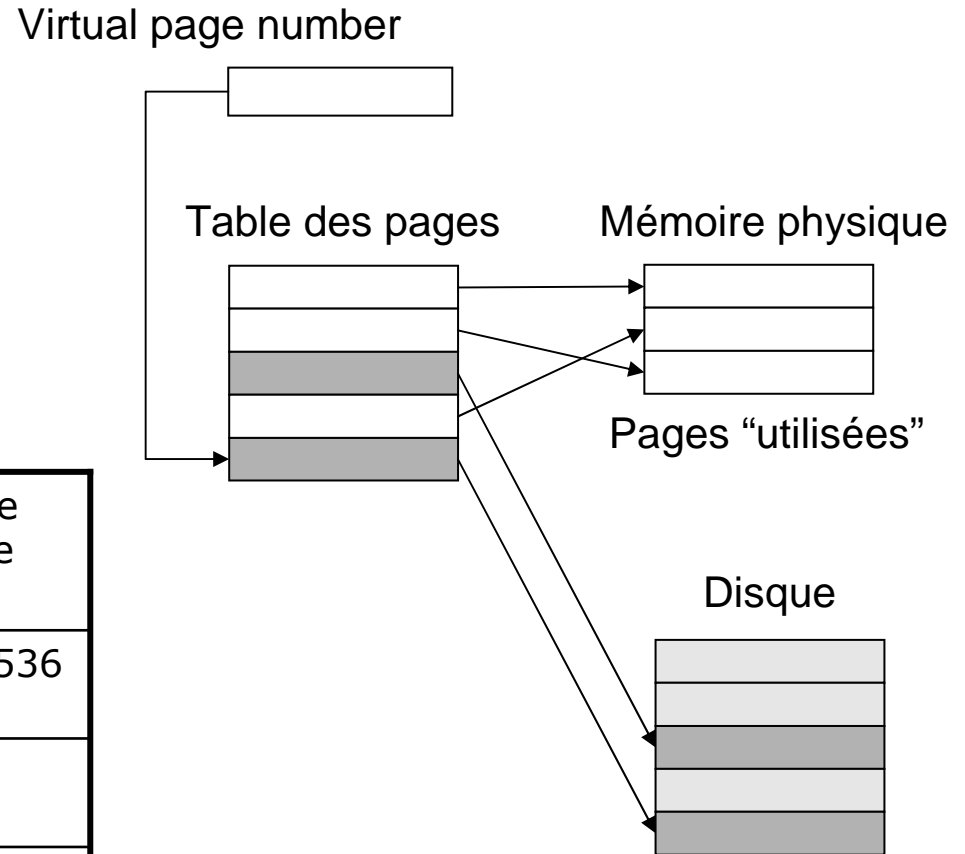


# Accélérer la translation d'adresses

Réduire la pénalité d'échecs

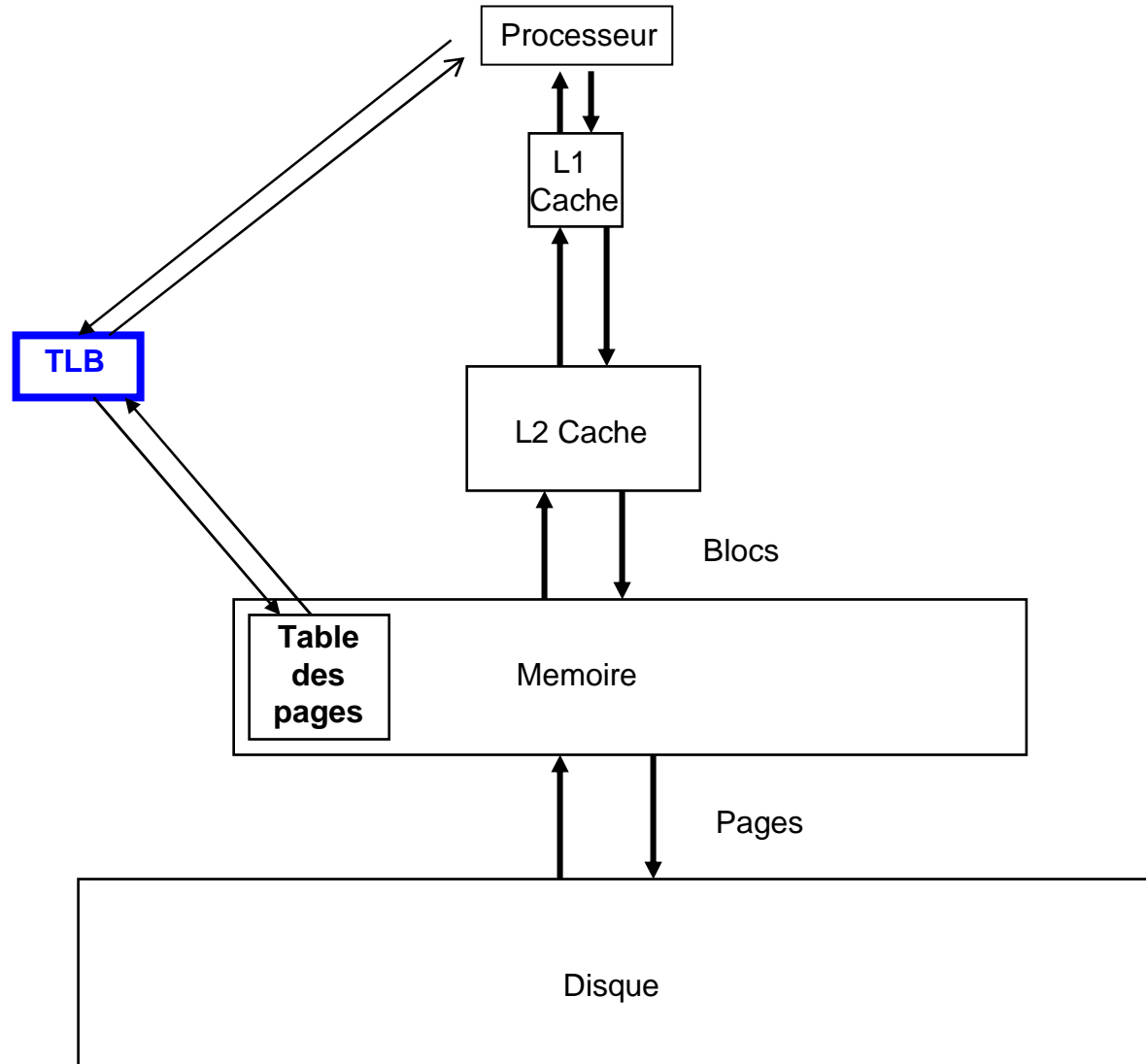
- Coût des accès mémoires pour la traduction d'adresses. Pénalité d'accès à la table des pages.

Paramètres	L1 cache	Mémoire virtuelle
Taille du bloc	16 ~ 128 B	4096~65,536 B
Temps d'accès	1~3 cycles	50~150 cycles
Pénalité d'échecs	8~150 cycles	1,000,000 ~ 10,000,000 cycles
Taux d'échecs	0.1~10%	0.00001~0.001%



# Hiérarchie mémoire avec TLB

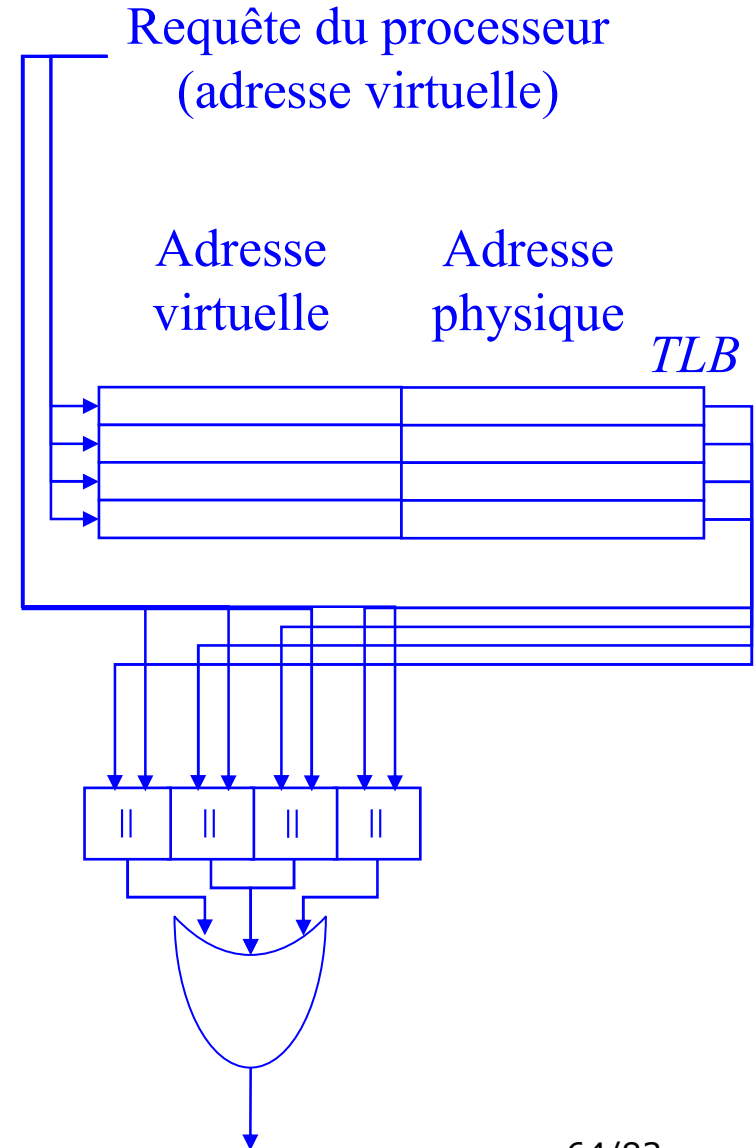
- Pour éviter les accès mémoires pour la traduction d'adresses (i.e. accélérer la traduction d'adresses), on utilise un cache spécifique pour les descripteurs de page : *Translation Lookaside Buffer* (tampon de traduction anticipée).
- Le TLB contient les quelques traductions de pages récemment accédées.



# Accélérer la translation d'adresses

Réduire la pénalité d'échecs

- TLB : similaire à un cache instructions (seulement en lecture) : le tag contient un numéro de page virtuelle, la partie donnée contient un numéro de page physique.
- TLB = cache classique (habituellement petit, 128 à 256 entrées), souvent complètement associatif.



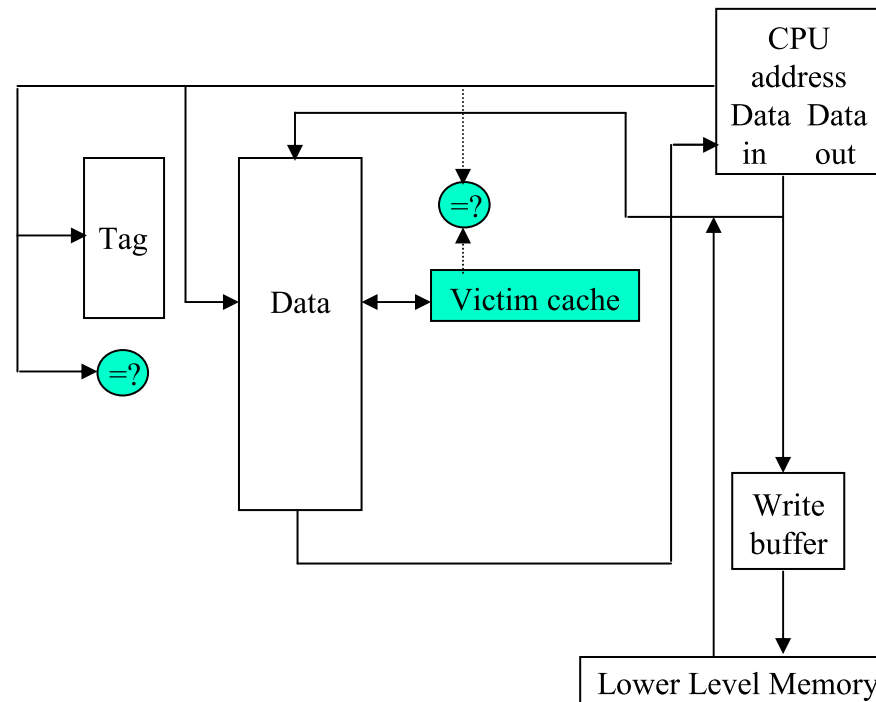


# Quelques techniques pour réduire le temps d'accès

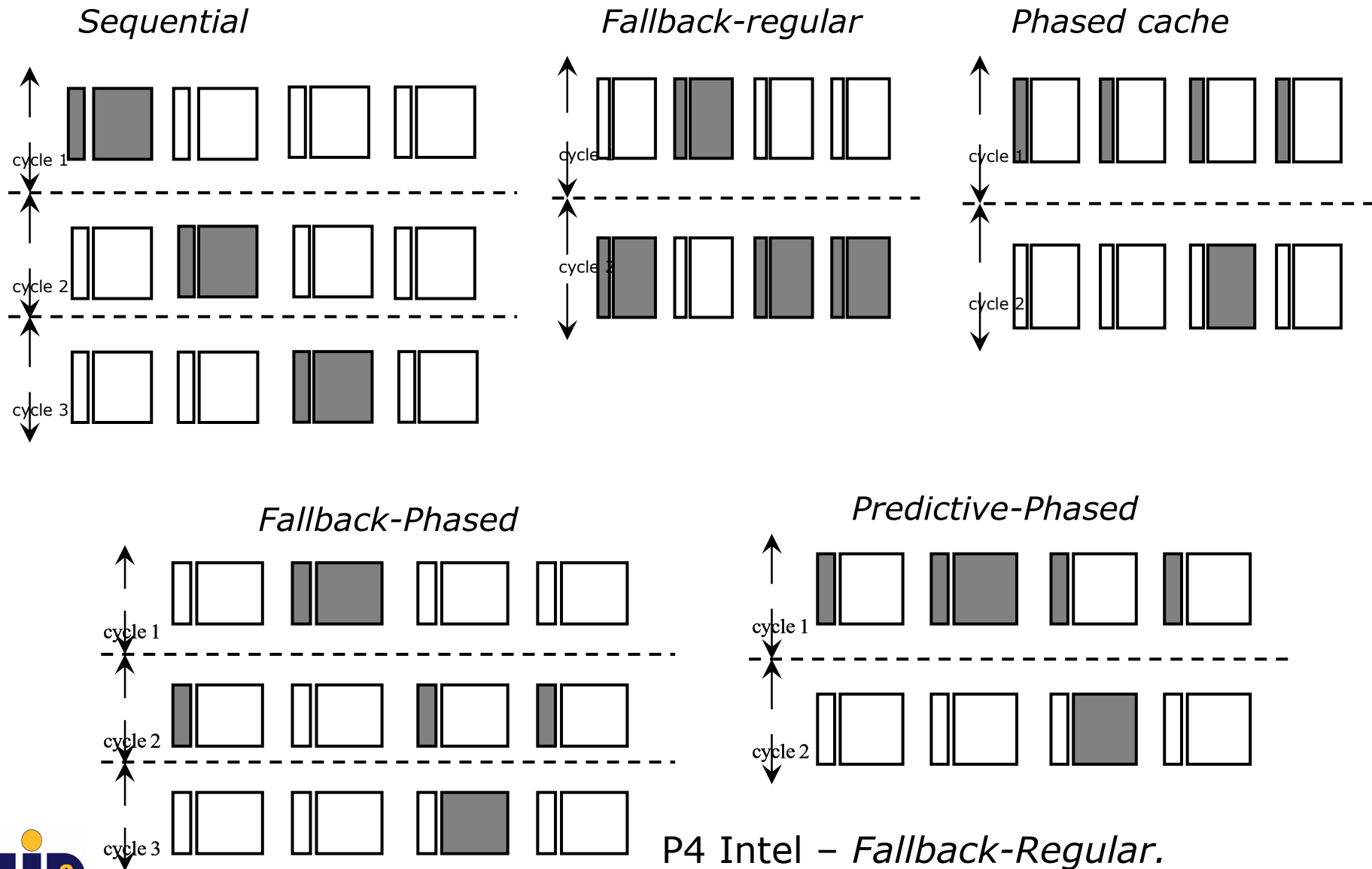
- Temps d'accès :
  - Temps d'accès à un niveau de cache donné,
  - Temps pour déterminer si c'est un succès ou un échec.
  - Taille pipeline –délai dépendances.
  
  - Directement lié à la fréquence,
  - Augmente avec la taille du cache,
  - Augmente avec l'associativité.
- Caches petits et simples,
- *Victim cache*,
- Translation d'adresses,
- ...

# Le Victim Cache

- Réduction des *conflicts misses* et du **temps d'accès au cache**.
- Ajout d'un buffer permettant de stocker les données évincées du cache.
- Jouppi [1990]: "4-entry victim cache removed 20% to 95% of conflicts for a 4-KB direct-mapped data cache".
- *Victim cache* « original » dans HP PA7200 (petit cache, fully-associative).
- Ancêtre du cache L2 on-chip...

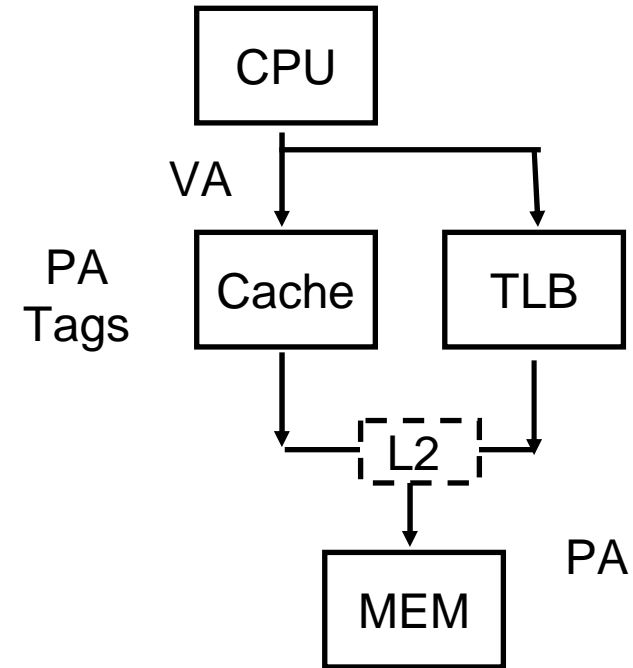
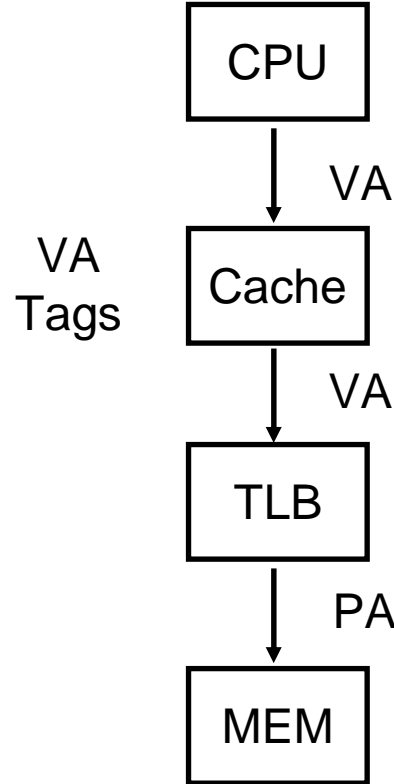
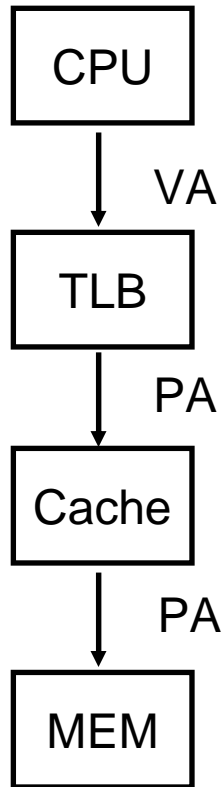


# Différentes politiques d'accès au cache



P4 Intel – *Fallback-Regular*.  
En réalité pour consommation... 67/83

# Eviter la traduction d'adresses



2 VA = même PA  
D'où plusieurs entrées  
dans cache avec même  
adresse physique.  
Purge.

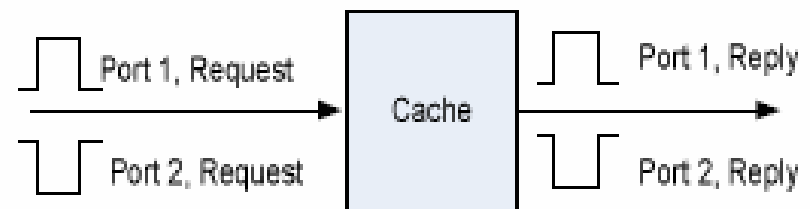
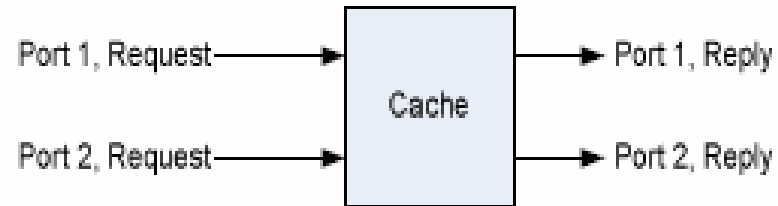
Taille limitée par  
index ou également  
problème de  
synonymes.

# Augmenter le débit d'instructions chargées

- Unité de chargement doit assurer un débit suffisant pour ILP :
  - Prédiction de branchement.
  - Préchargement d'instructions :
    - Chargement de plus d'1 instruction à la fois – combiné à la prédiction de branchement pour précharger des instructions utiles.
  - Chargement de plusieurs instructions (cache multi-ports) :
    - Multi-ports,
    - *Overclocking*,
    - Cache multiple,
    - Bancs.
  - Une combinaison : trace cache.

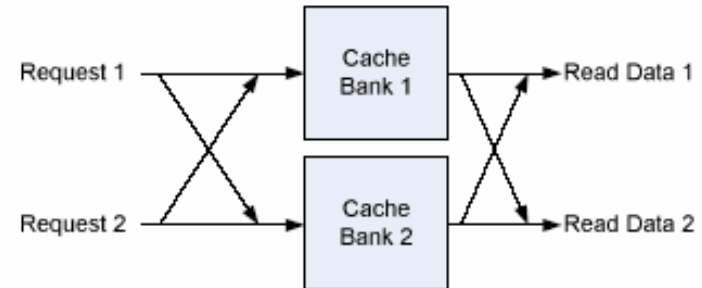
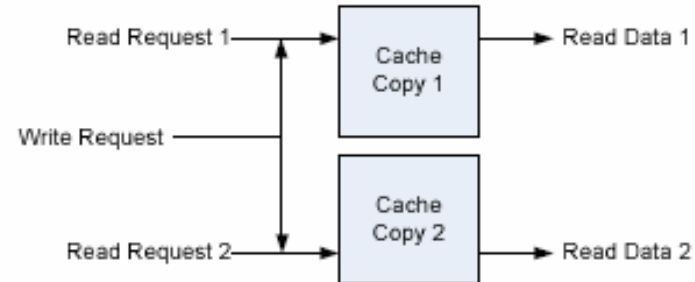
# Cache multi-ports

- Multi-ports : 2 ports (tag + data).
  - Augmentation importante de la taille du cache et du temps de succès.
- *Overclocking* : clock du cache 2 fois plus rapide que processeur. Un accès par  $\frac{1}{2}$  clock.



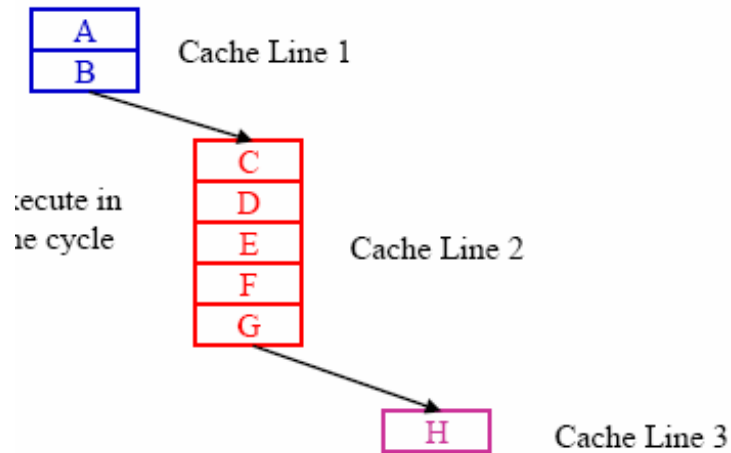
# Cache multi-ports

- Cache multiple (2 load / 1 store).
  - Taille \* 2.
- Partition des adresses en plusieurs bancs :
  - Conflits potentiels entre les bancs (pas d'accès multiples).



# Trace cache

- Cache qui capture les séquences dynamiques d'instructions (décodées). Contient donc des segments d'instructions contenant des branchements, potentiellement pris.

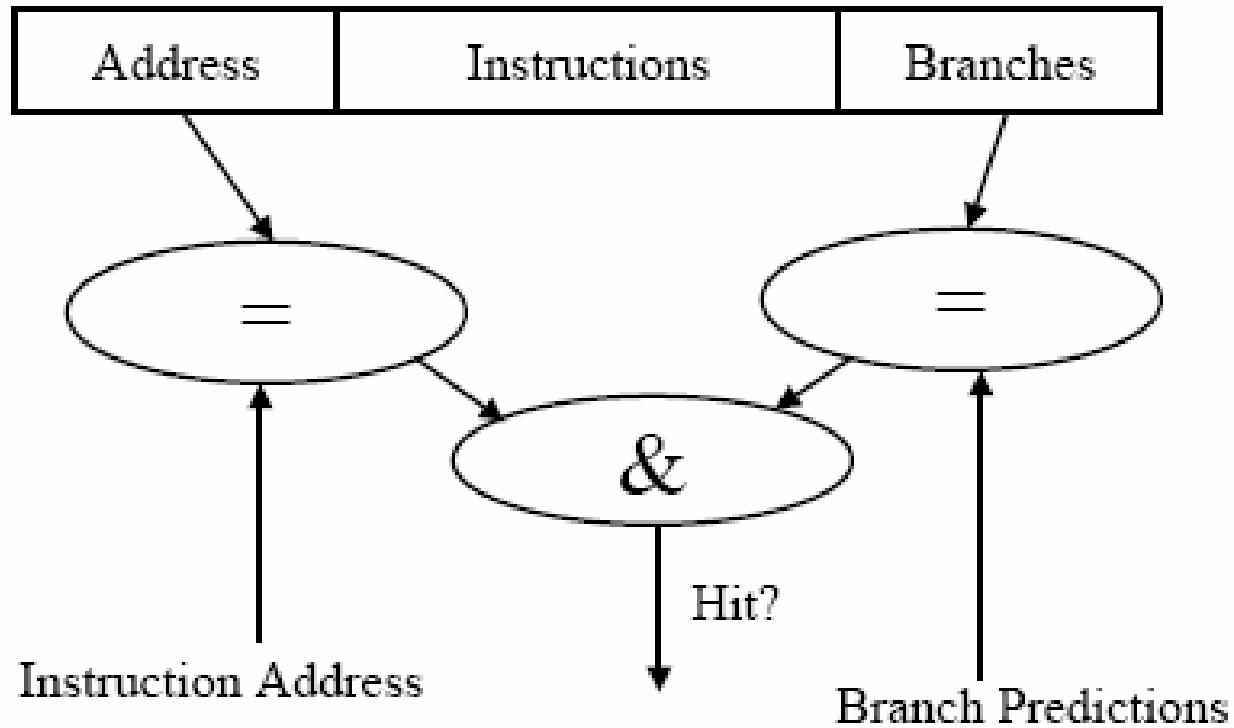


Start Address	Instructions								Branches
	A	B	C	D	E	F	G	H	1,1,1



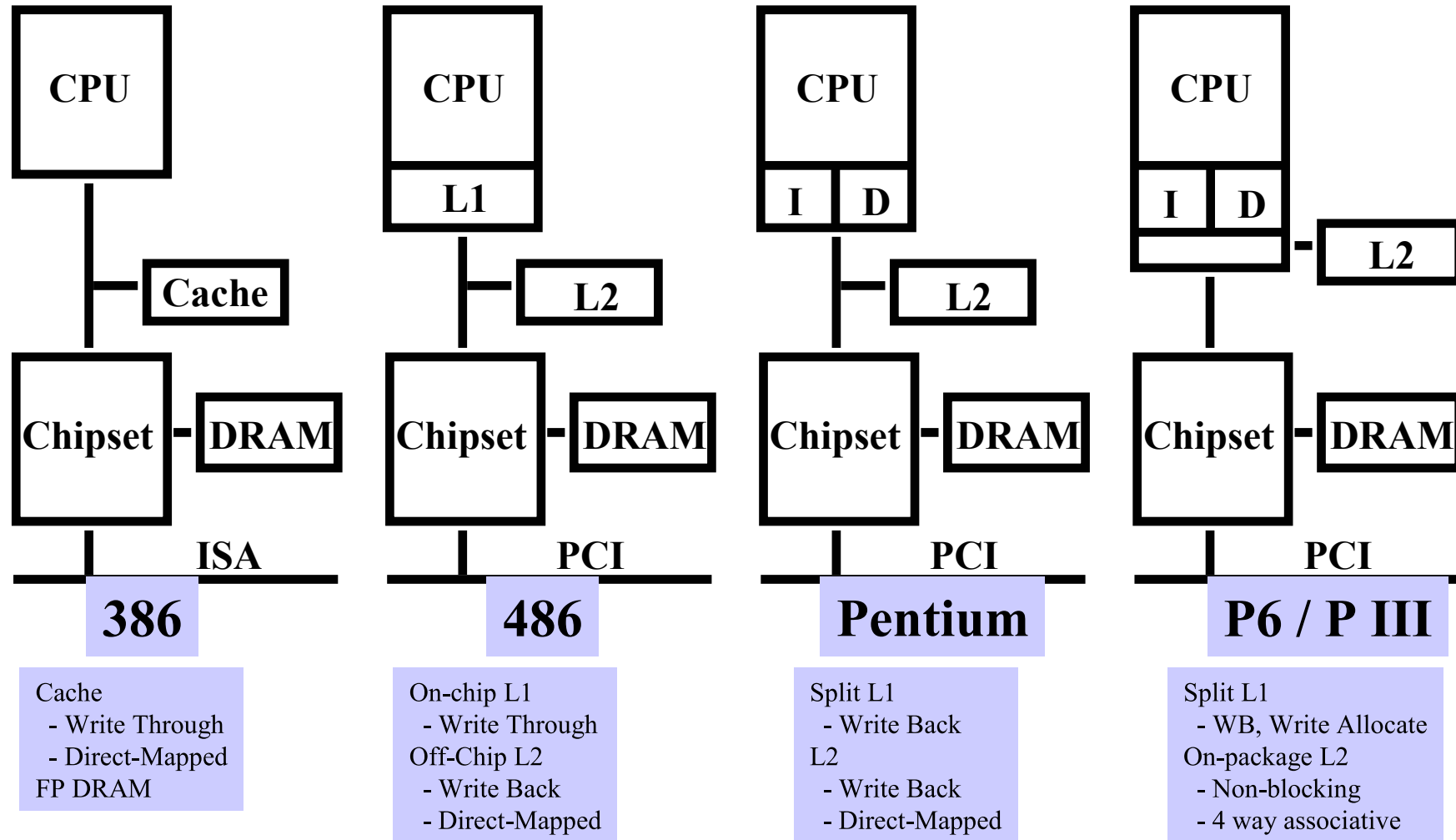
# Trace cache (suite)

- On compare l'adresse et la prédiction.

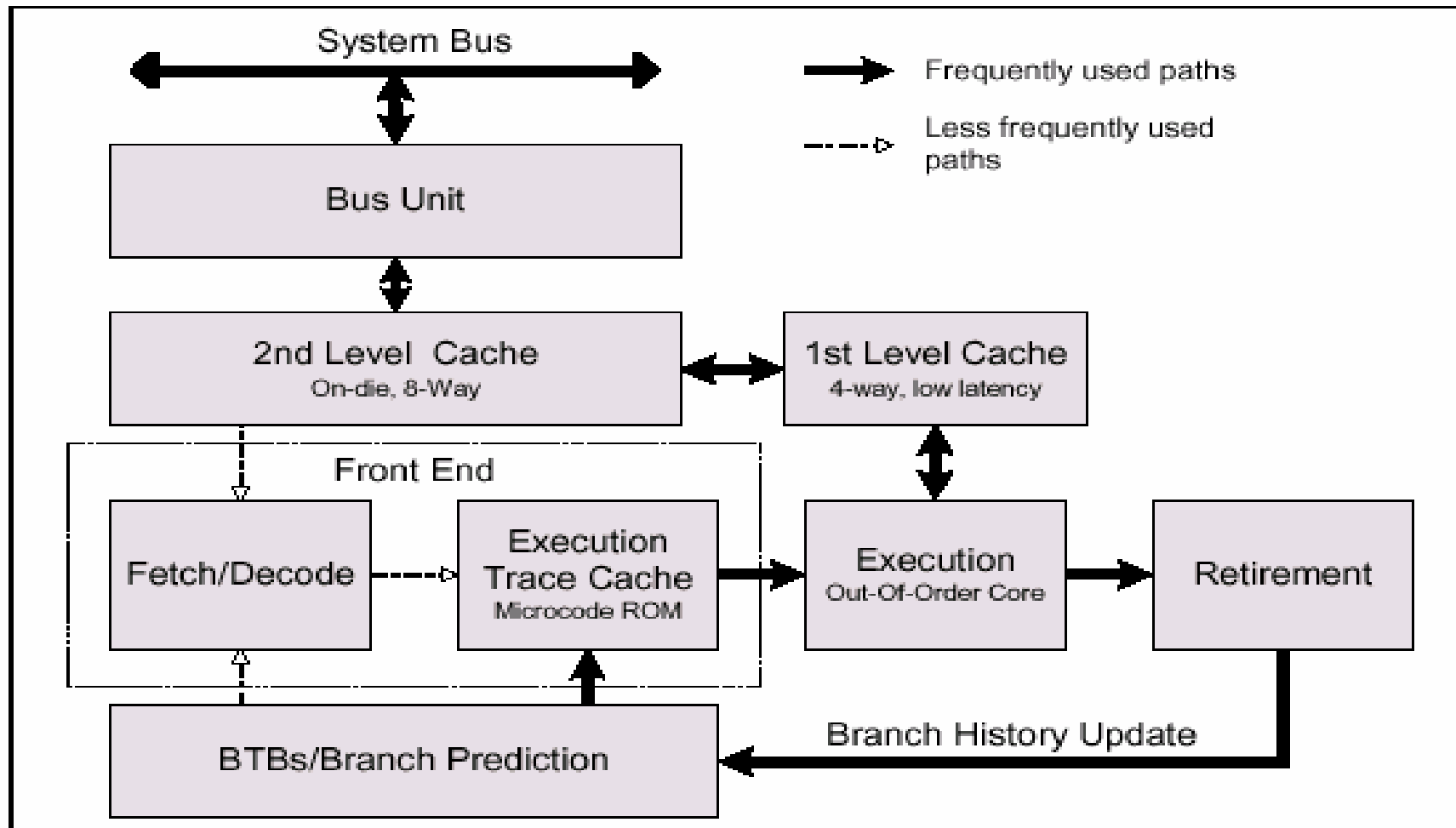


# Un exemple de hiérarchie mémoire

# Evolution



# Processeur Pentium 4



# Hiérarchie mémoire

- L1 instruction cache (Trace Cache),
- L1 data cache : non-bloquant (supporte 4 échecs en suspend), 1 Load et 1 Store par cycle. Bloc de 64 octets.
- L2 cache unifié : non-bloquant, 1 load et 1 store par cycle, bus entre le L1 et le L2 = 256 bits, préchargement matériel (256 bytes à la suite d'un accès et 8 flots indépendants simultanément). Bloc de 128 octets. Préchargement 128 octets (+variantes).
- Politique de remplacement = pseudo-LRU.

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency, Integer/floating-point (clocks)	Write Update Policy
First	8KB	4	64	2/6	write through
TC	12K $\mu$ ops	8	N/A	N/A	N/A
Second	256KB	8	128 <sup>1</sup>	7/7	write back

<sup>1</sup> Two sectors per line, 64 bytes per sector

# D'autres techniques pour accéder plus rapidement aux données

## Instruction flow :

prédiction  
de branchement  
sophistiquée, trace cache,  
réutiliser les instructions.

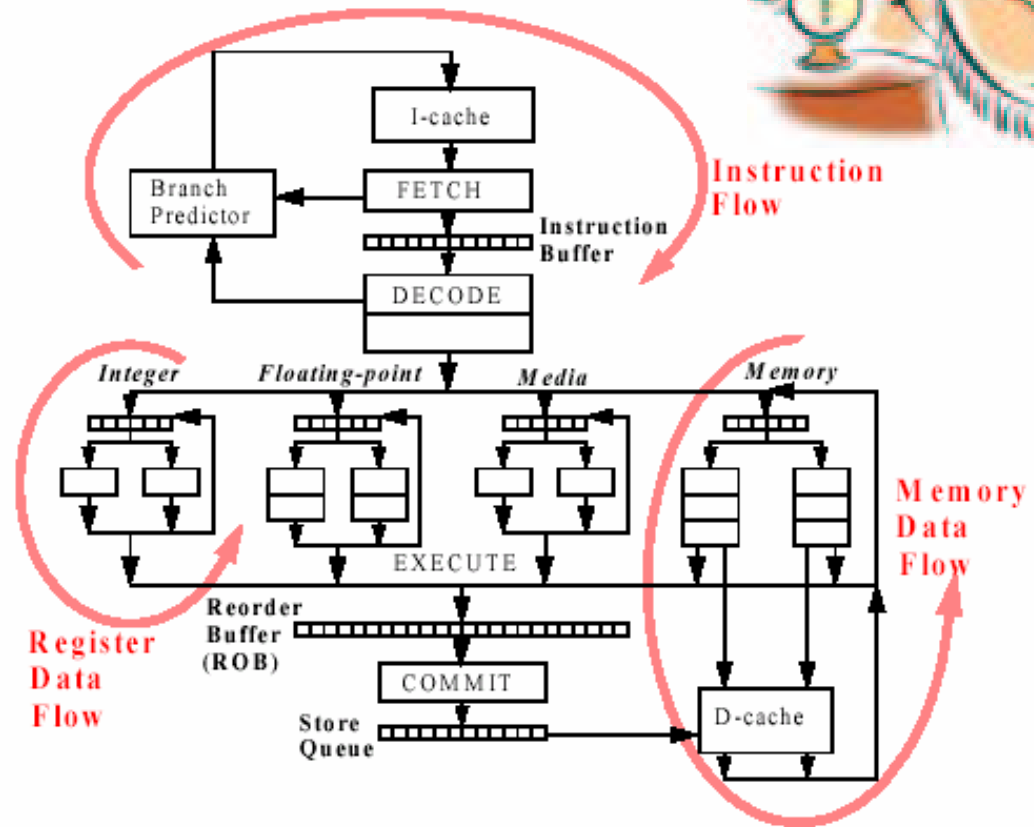
## Register data flow :

renommage  
de registres spéculatif,  
prédiction de valeur.

## Memory data flow :

charger en avance  
(préchargement).  
Prédiction d'adresses, de  
dépendances et de  
valeurs – **masquer  
latences locales.**

Que peut-on  
prédire ?

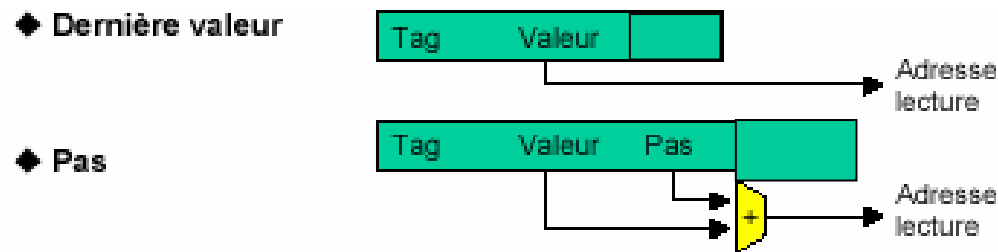


# Memory data flow : prédiction de dépendances

- Exécution des accès mémoire dans le désordre.
- Problème : est-il nécessaire d'attendre que toutes les adresses antérieures soient connues ? (Load suivant un Store en attente).
- Solution : dépassement avec prédiction et transmission :
  - Prédications :
    - Prédiction « aveugle » : une lecture est supposée indépendante de toutes les écritures en attente (systématique).
    - Prédiction par bit d'attente : à chaque instruction du cache est associé un bit d'attente. Ce bit est mis à 1 si dépendance observée à l'exécution précédente (remis à 0 à intervalles réguliers, car les adresses changent).
  - Les instructions qui dépendent du load sont exécutées avec la valeur chargée spéculativement. Nécessité d'un mécanisme pour relancer les instructions fautives en cas d'erreur de prédiction = lorsqu'une écriture reçoit son adresse.

# Memory data flow : prédiction d'adresse

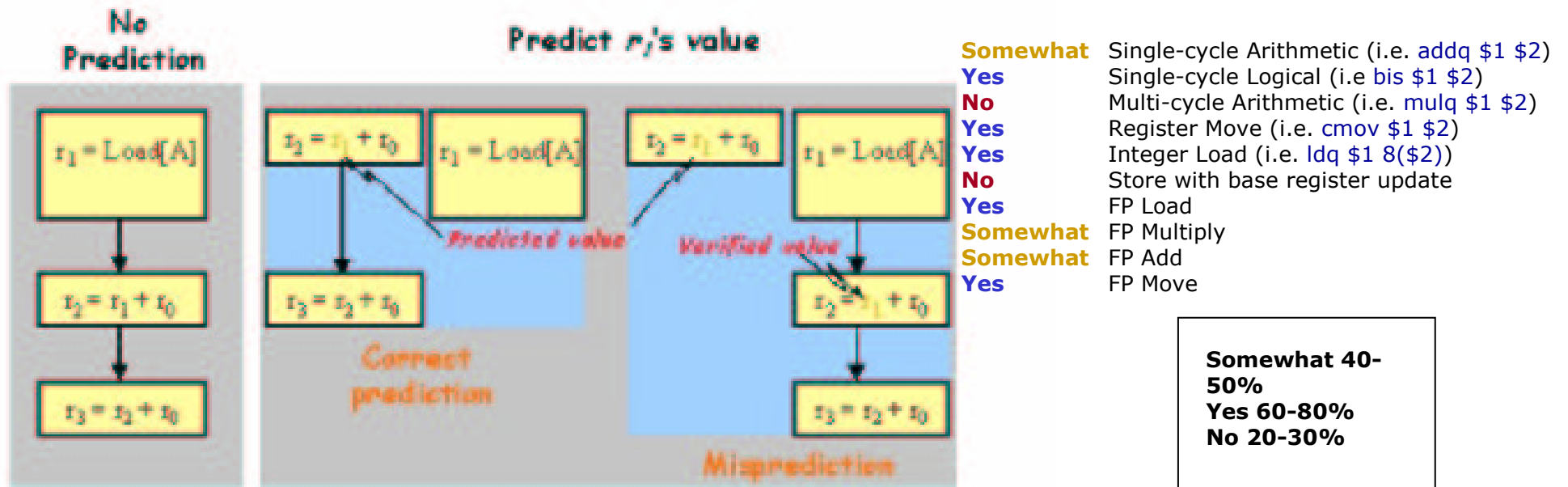
- Prédiction pour ne pas attendre le calcul de l'adresse.



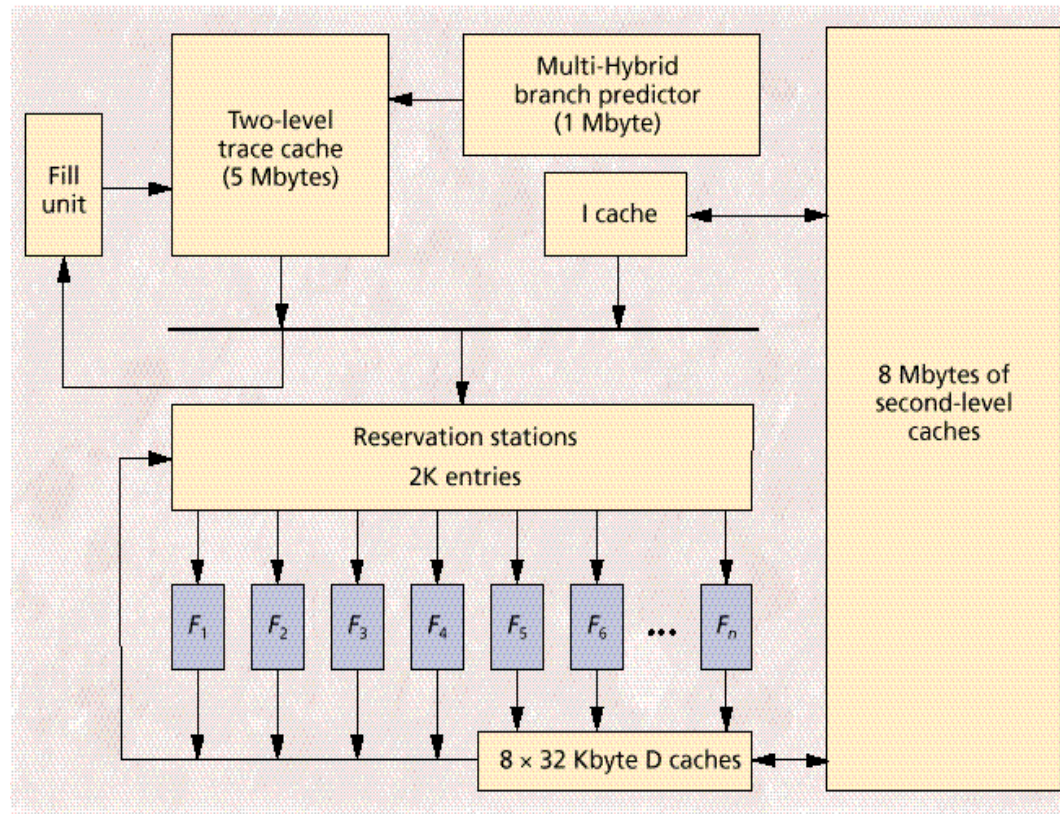


# Memory (register) data flow : prédire les valeurs

- Extension de la prédiction d'adresse d'une lecture.
- Prédiction de la valeur produite par une instruction : dernière valeur, pas, autres...
- Les instructions dépendantes peuvent être exécutées plus tôt avec une valeur spéculative.



# Dimensionner plus largement les processeurs superscalaires (budget de 1 Milliard de transistors – ~2010)



- ❑ Prédicteur de branchement performant de grande taille.
- ❑ Un cache trace large.
- ❑ Plusieurs instructions exécutables en parallèle (16 ou 32 instructions par cycle).
- ❑ Un grand nombre de stations de réservation (2000).
- ❑ 24 à 48 unités fonctionnelles pipelinées.
- ❑ Un large cache de données on-chip.

# Mais aussi la consommation...

- Réduire la consommation des opérations qui consomment beaucoup et notamment :
  - réduire le nombre d'accès à la mémoire,
  - réduire la consommation des caches.
- Rapport consommation due aux accès mémoire / consommation due aux accès cache :
  - Rapport grand : se focaliser sur l'amélioration du taux de succès.
  - Rapport petit : se focaliser sur minimiser la consommation d'énergie des accès au cache.
- Obtenir plus de performance : augmenter la taille du cache, l'associativité et la taille de la ligne.
- Limiter la consommation : réduire la taille du cache et l'associativité. Mais aussi réduire le nombre d'accès à la mémoire.
- Compromis...

