Archi 3 : Architecture, Programmation et Compilation des processeurs embarqués Haute Performance

Bertrand Granado - Quentin Meunier - Lionel Lacassagne

Fonctionnement et Optimisation des Mémoires Cache

Quentin Meunier

Laboratoire d'Informatique de Paris 6
Équipe Alsoc
4 Place Jussieu, 75252 Paris, France

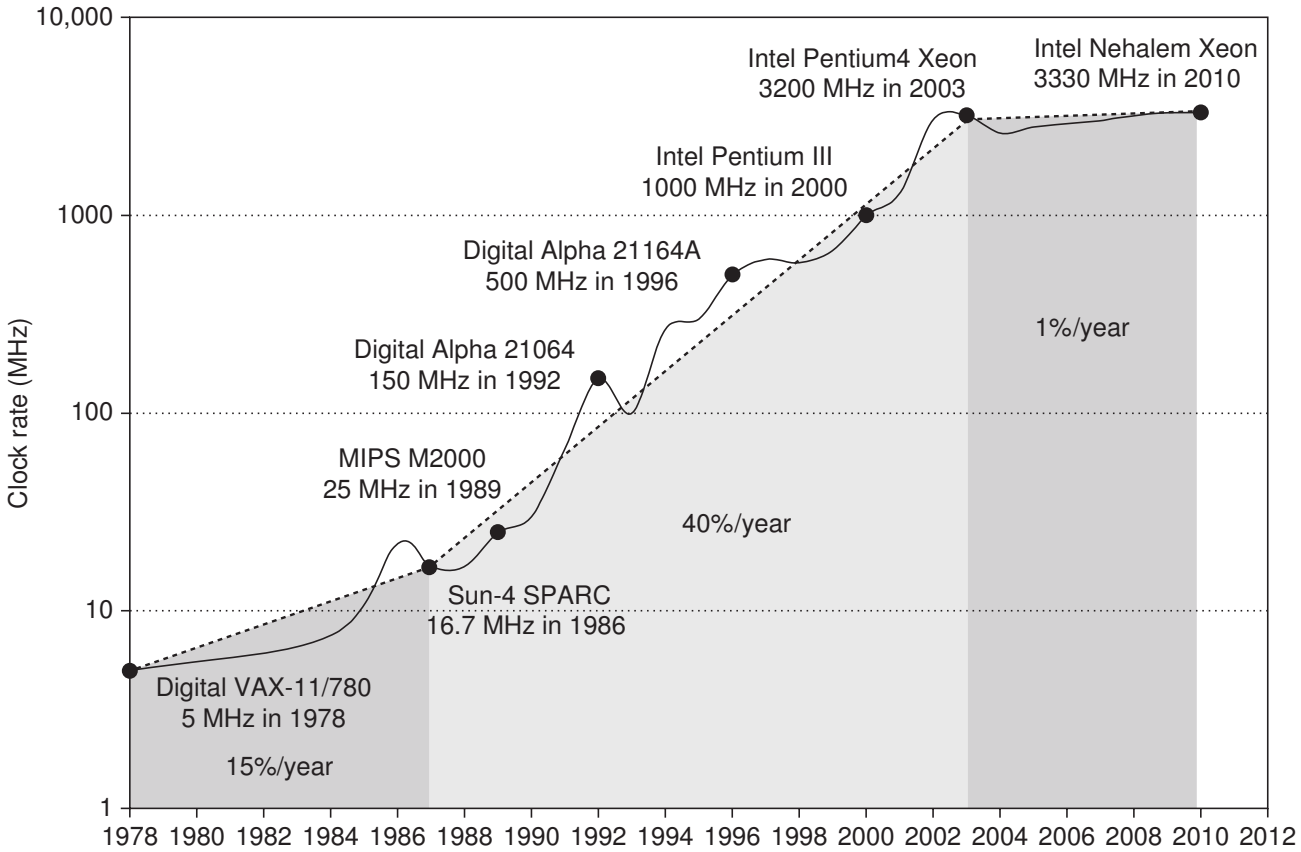
Septembre 2018

Références pour le cours

- *Computer Architecture : A Quantitative Approach*, 5e édition, John Hennessy et David Patterson, 2011
- *What Every Programmer Should Know About Memory*, Ulrich Drepper, 2007 (en accès libre)

Apport de l'architecture dans la performance

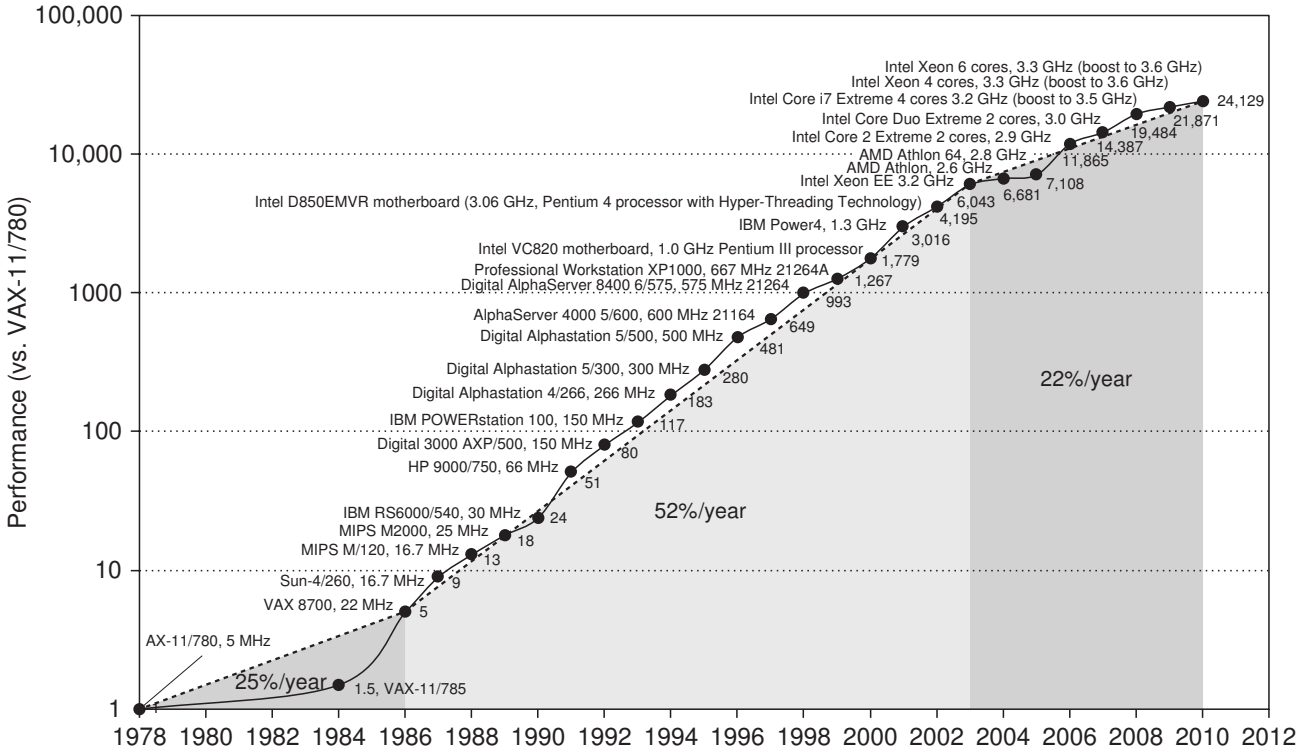
- Augmentation des performances des processeurs : combinaison entre technologie et (micro)architecture
- Evolutions technologiques :
  - Augmentation du nombre de transistors
  - Augmentation de la fréquence



"Computer Architecture" J. L. Hennessy, D. A. Patterson

Apport de l'architecture dans la performance

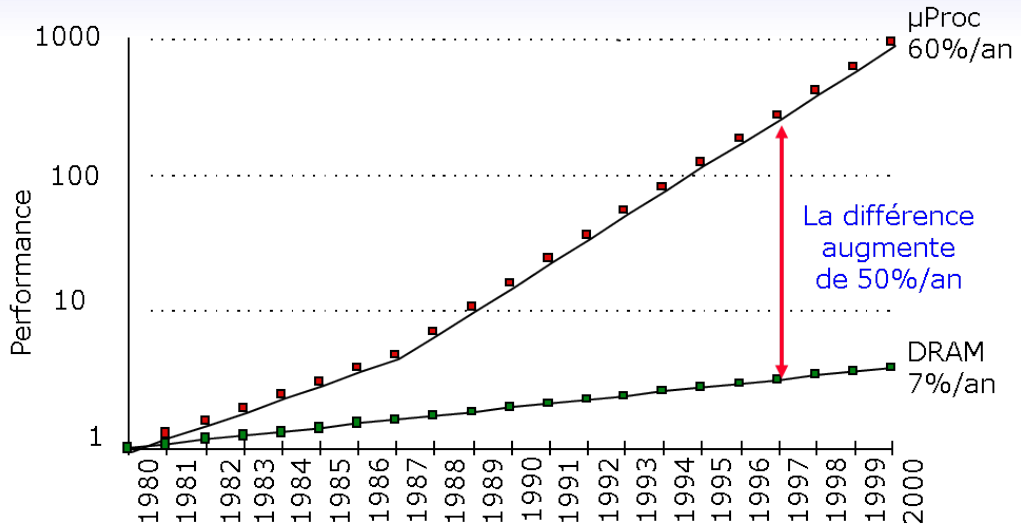
- Augmentation des performances des processeurs : combinaison entre technologie et (micro)architecture
- Evolutions en architecture :
  - Traduire l'augmentation des transistors en augmentation de performance



"Computer Architecture" J. L. Hennessy, D. A. Patterson

- La fréquence joue un rôle dans l'augmentation des performances mais n'est pas la seule responsable

Un problème limitant la performance : le problème de la latence mémoire



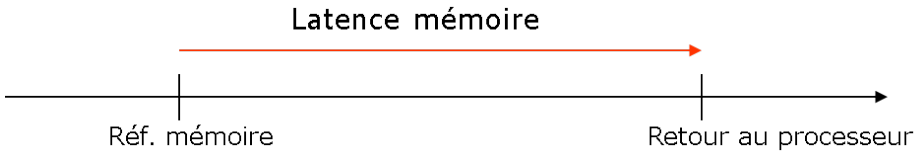
Problématique

- Temps de cycle processeur $<$ temps d'accès mémoire
- Requête mémoire du processeur = processeur inactif plusieurs cycles

1986 : temps de cycle processeur $\sim$ 120 ns, temps d'accès à la mémoire $\sim$ 140 ns } 1

1996 : temps de cycle processeur $\sim$ 4 ns, temps d'accès à la mémoire $\sim$ 60 ns } 20

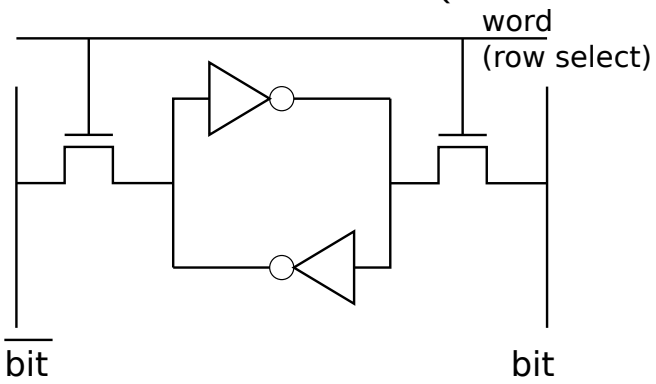
2002 : temps de cycle processeur $\sim$ 0.6 ns, temps d'accès à la mémoire $\sim$ 50 ns } 100



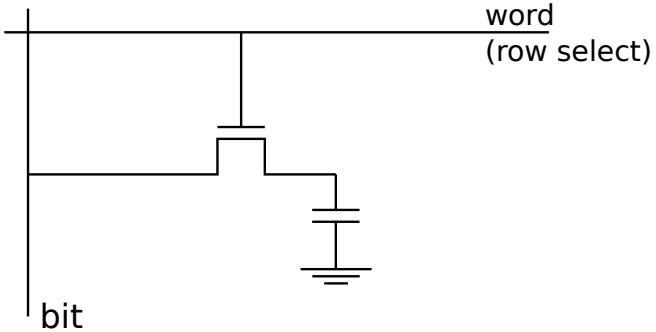
Masquer la latence mémoire : mémoire cache

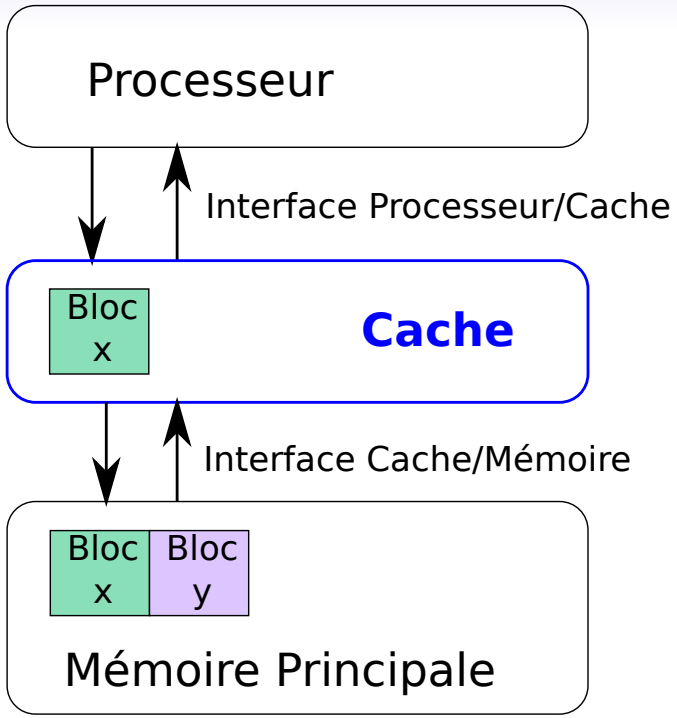
- Temps de cycle processeur $\ll$ temps d'accès mémoire
- Requête mémoire du processeur = processeur inactif plusieurs cycles

Cellule mémoire SRAM (6 transistors)



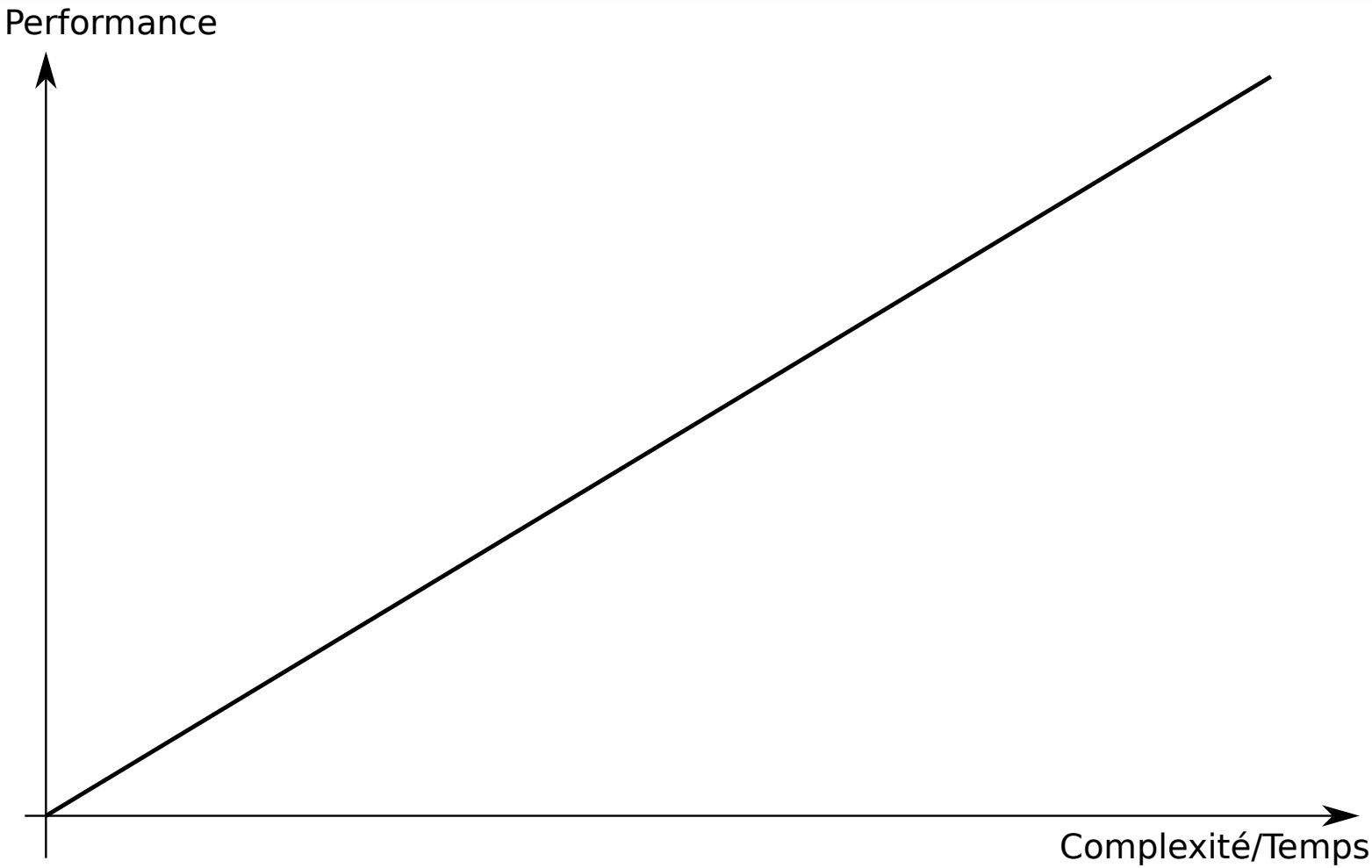
Cellule mémoire DRAM (1 transistor)



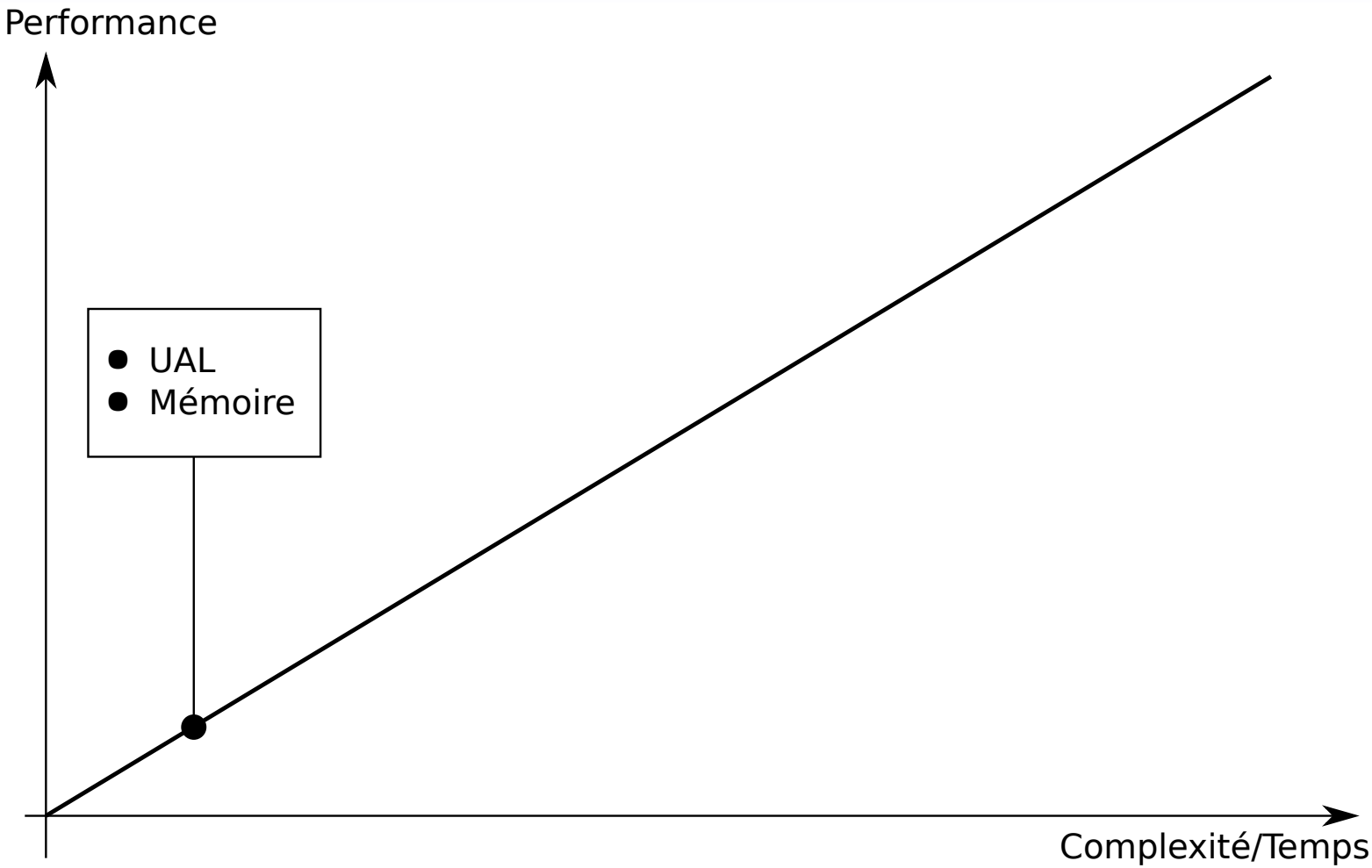


Technologie mémoire	Temps d'accès typique	$ par Go 2004	$ par Go 2015
SRAM	0.5 – 5 ns	4K – 10K	150
DRAM	50 – 70 ns	150 – 200	5 – 8
Disque magnétique	5.10^6 – 20.10^6 ns	0.50 – 2	0.05

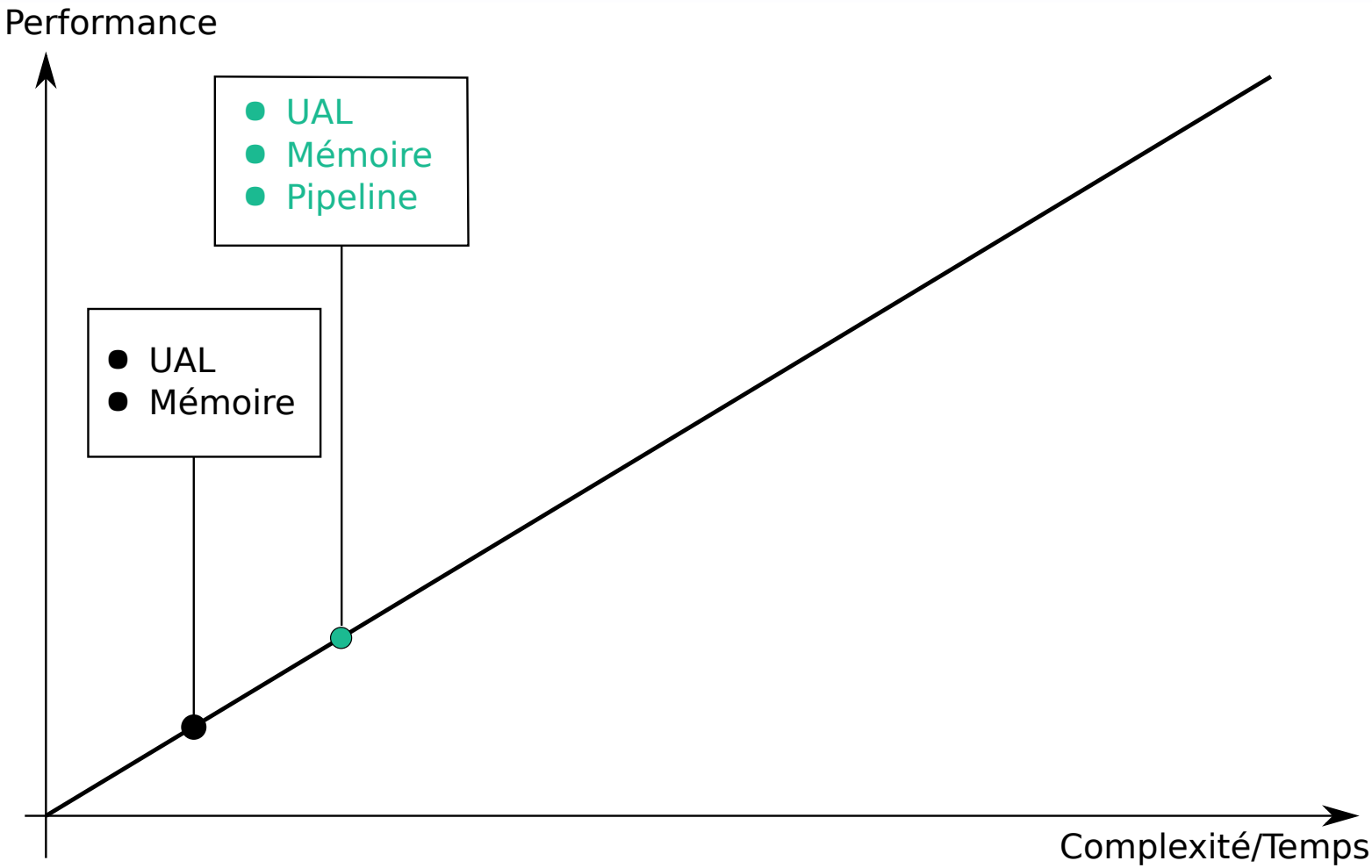
Evolution architecture haute-performance vs. cache



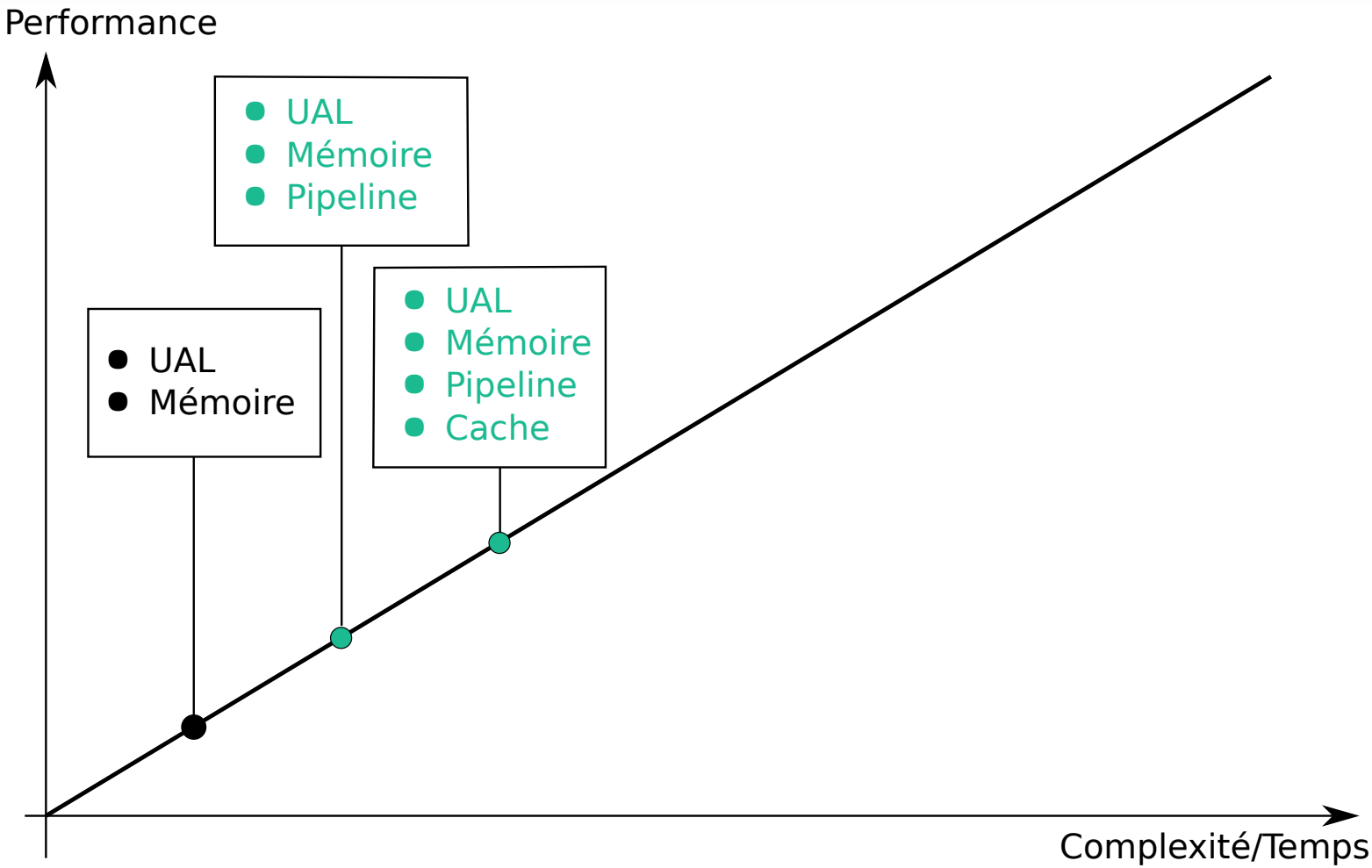
Evolution architecture haute-performance vs. cache



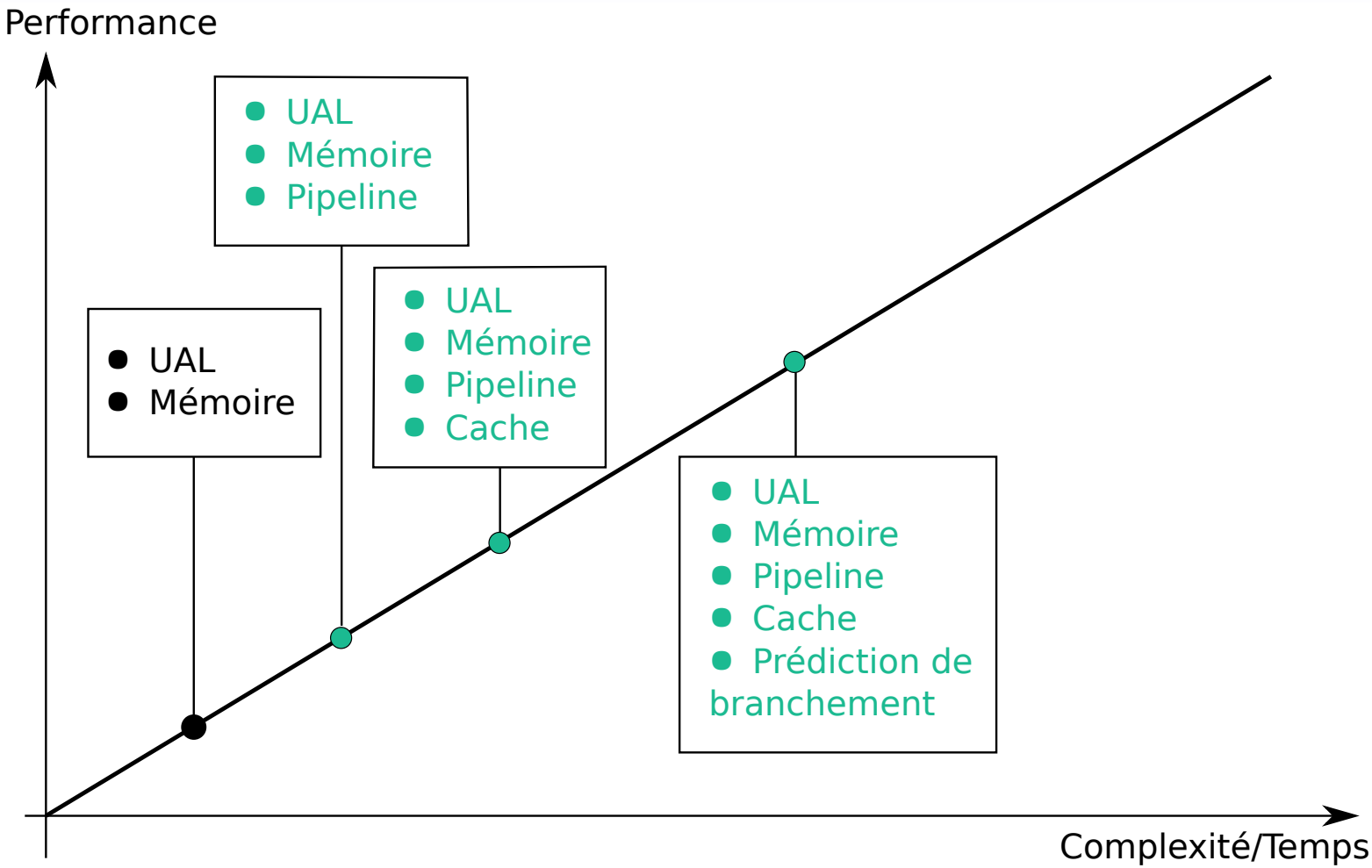
Evolution architecture haute-performance vs. cache



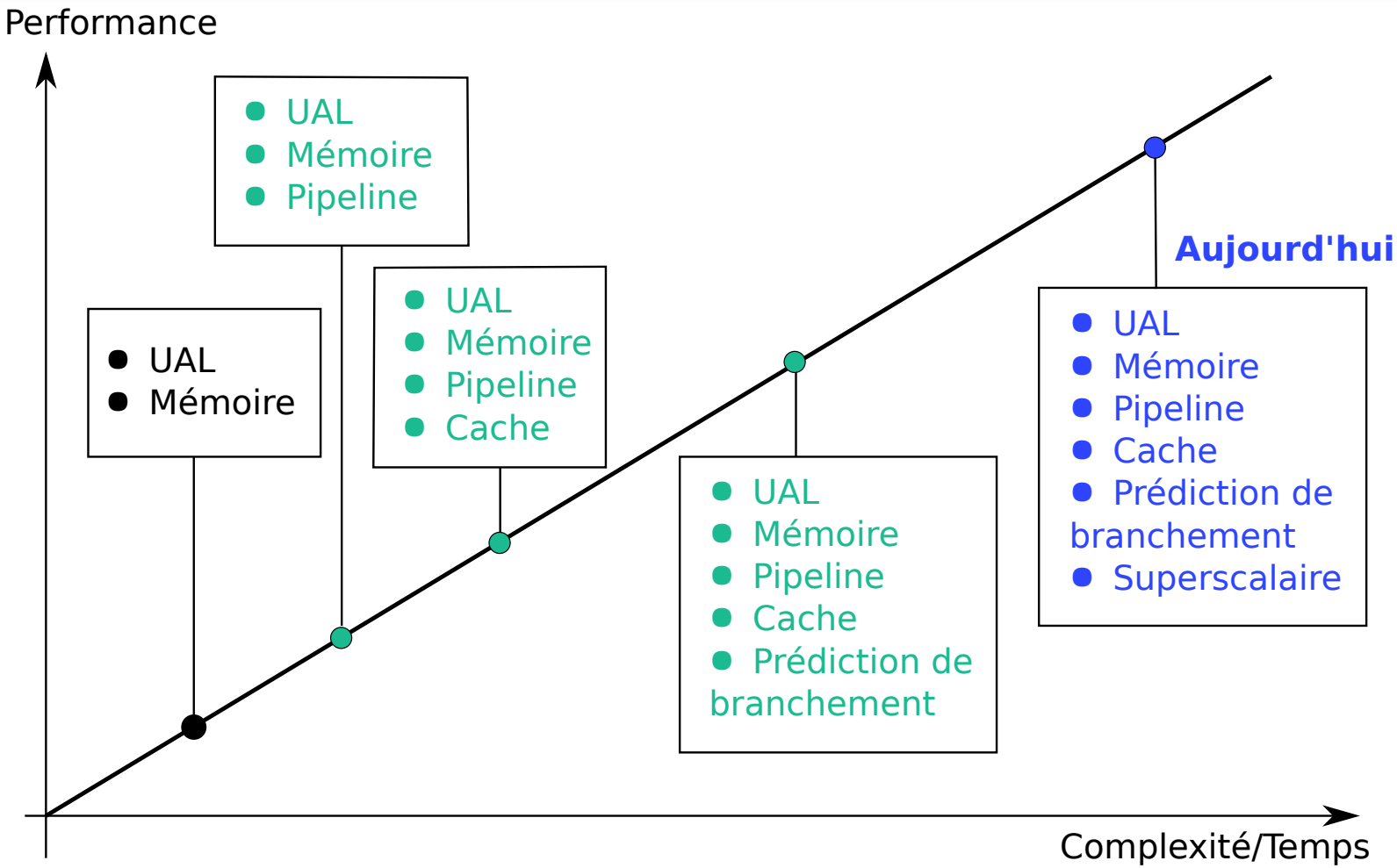
Evolution architecture haute-performance vs. cache



Evolution architecture haute-performance vs. cache



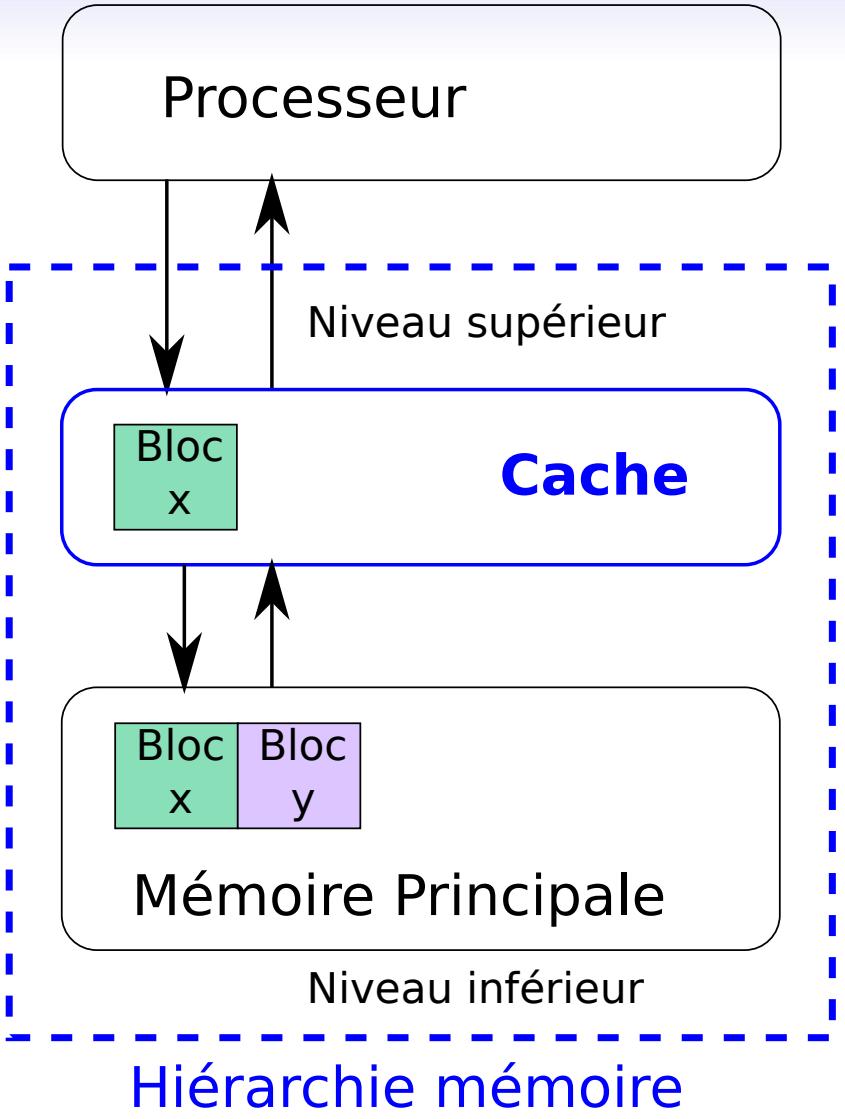
Evolution architecture haute-performance vs. cache



Comment ça marche ?

Principes

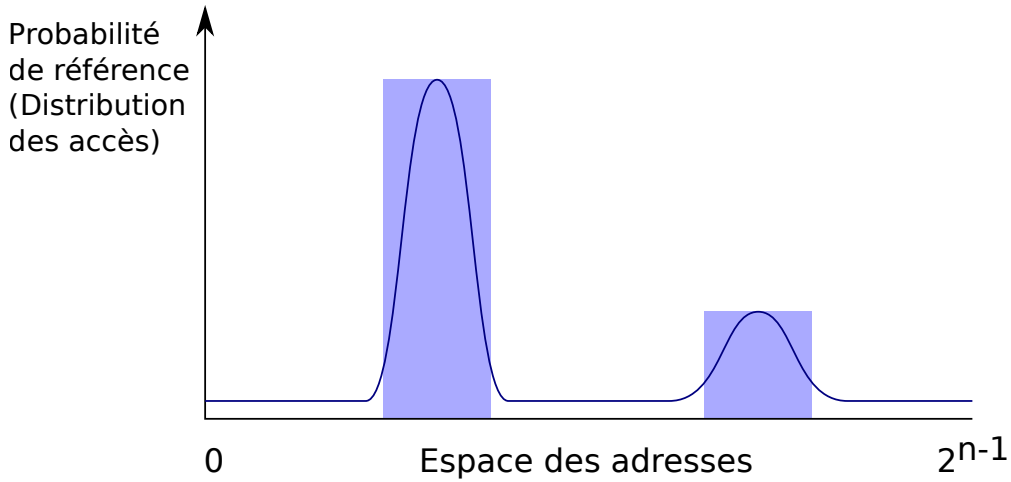
- Le processeur envoie ses requêtes mémoire au cache
  - Donnée dans le cache : **succès** (**hit**)
  - Donnée hors du cache : **échec** (**miss**)
- Temps de succès $\ll$ pénalité d'echec
- Temps de succès = temps d'accès + temps pour déterminer si succès ou échec
- Notion de **Hiérarchie Mémoire**



Pourquoi ça marche ?

Principe de localité

- A un instant donné, un programme accède à une portion relativement petite de l'espace d'adressage

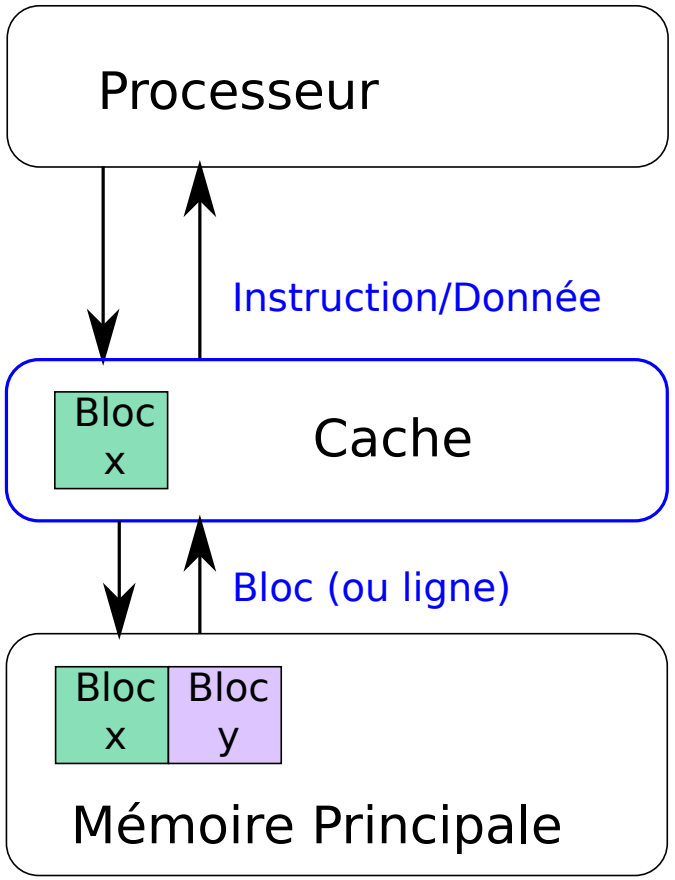


- **Localité temporelle** : Si l'adresse A est référencée à T, alors forte probabilité de référencer A à $T + t$, avec t petit (réutilisation).
- **Localité spatiale** : Si l'adresse A est référencée à T, alors forte probabilité de référencer $A + a$ à $T + t$, avec a petit et t petit

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



Localité des données et des instructions

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

Données

```
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] : propriétés de localités temporelle et spatiale

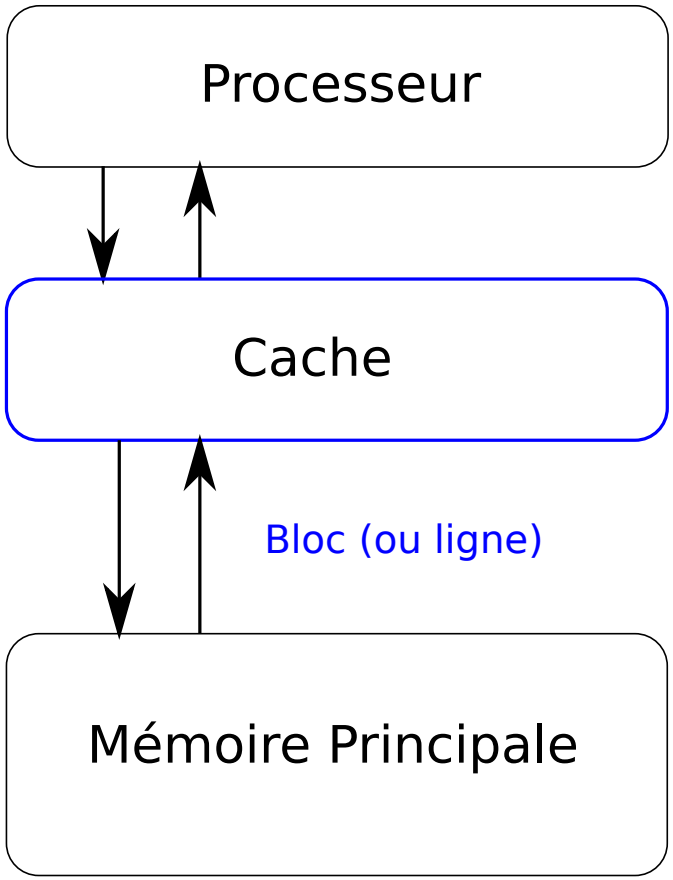
Instructions

```
      ...
loop: lw   $5, 4($16)
      addu $5, $5, 5
      sw   $5, 8($16)
      addu $5, $5, $6
      addu $4, $5, $17
      bne  $4, $8, 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 et politique de placement
- Politique d'écriture
- Politique d'allocation

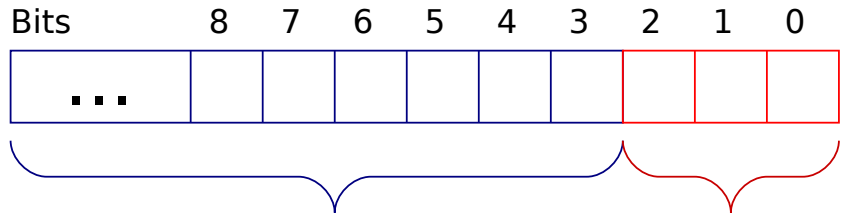


Bloc ou ligne de cache

- 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
  - La partie basse de l'adresse varie, elle indique l'**offset** dans la ligne
- Rappel : une adresse référence un octet

Exemple :

- Adresse sur 16 bits
- Bloc de 8 octets



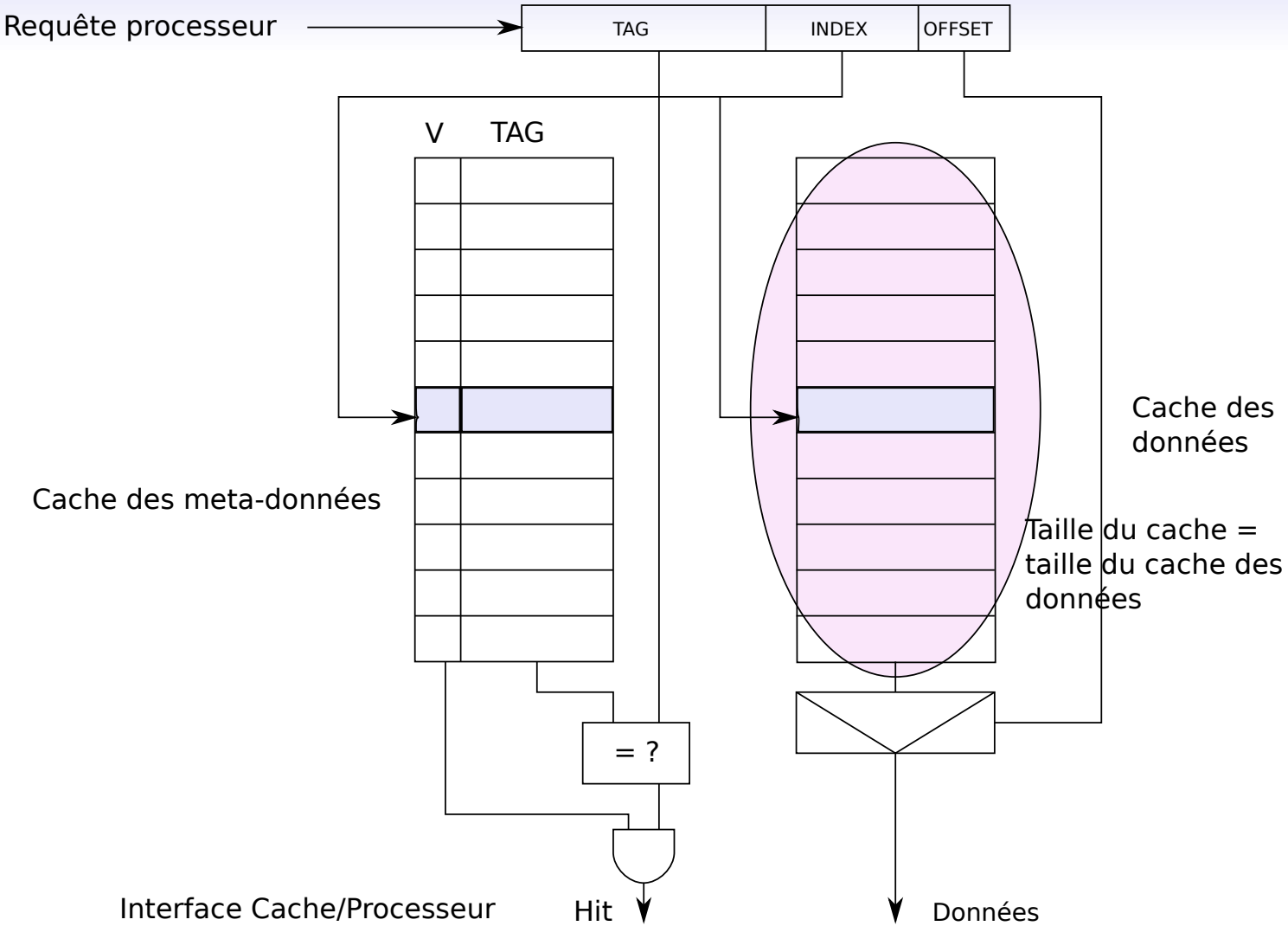
Partie haute de l'adresse : Adresse de la ligne (ou numéro du bloc)

Partie basse de l'adresse

000...010100000
000...010100111
Même ligne de cache

000...010101000
Lignes distinctes, adresses consécutives

Structure générale d'un cache



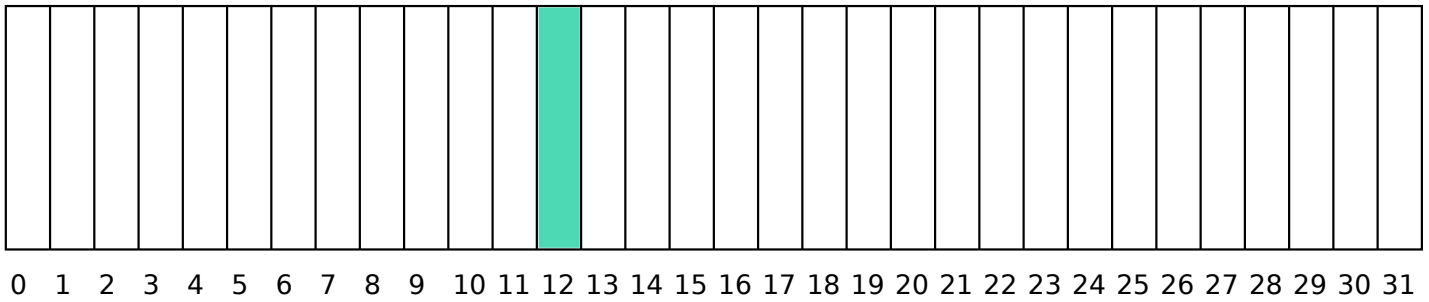
Placement des données dans le cache

- 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 une mémoire locale)
  - Le fonctionnement du cache est transparent pour le programmeur
- $\Rightarrow$ Politique de placement simple

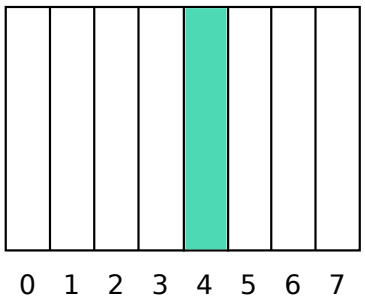
Différentes organisations de cache

- Où placer une ligne de la mémoire principale dans le cache ?
- Exemple : lignes de n octets, adresses de $5 + \log_2(n)$ bits, cache de 8 lignes

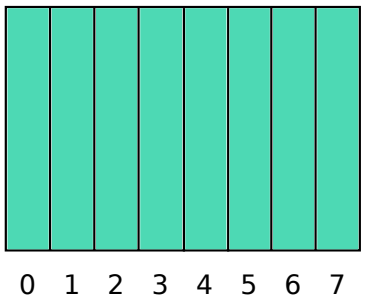
Numéro de bloc en mémoire



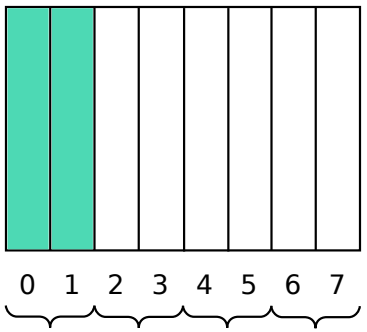
Direct Mapped



Fully Associative



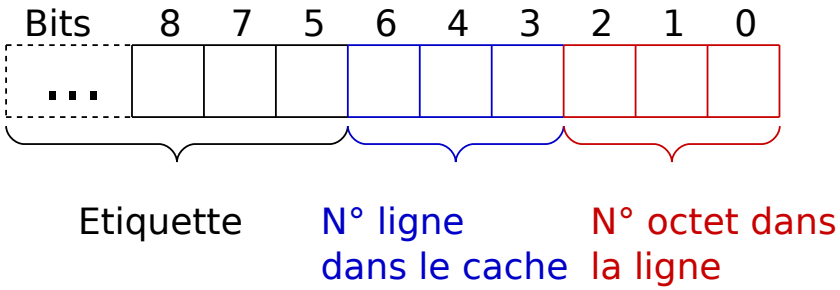
Set Associative (2-way)

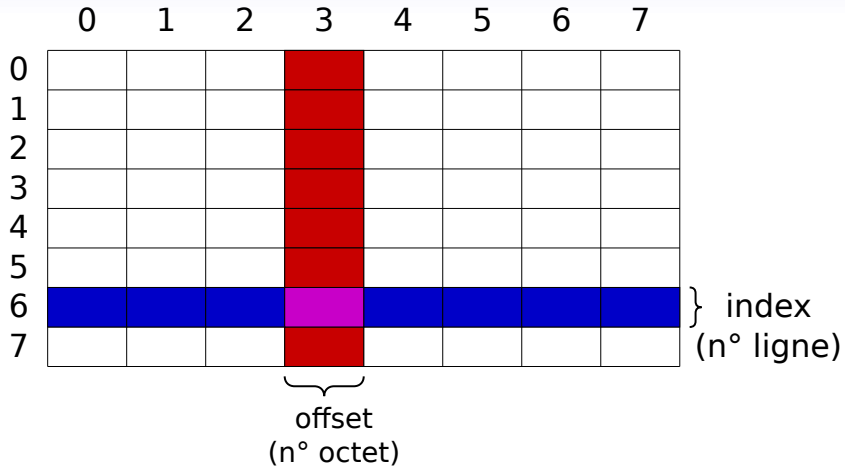


Numéro de bloc dans le cache

Placement des données – Cache à correspondance directe

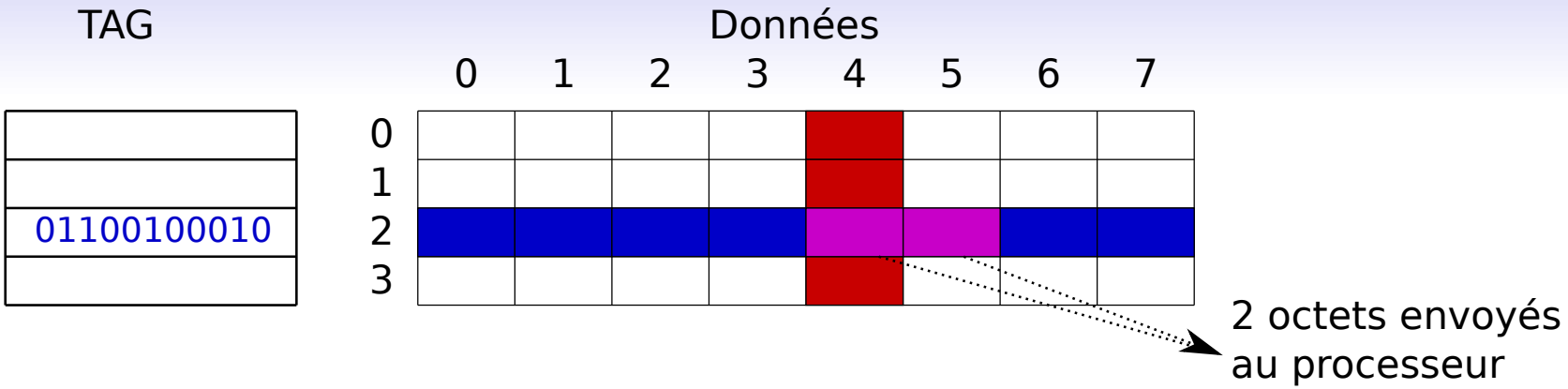
- L'emplacement d'une donnée est déterminé selon une fonction simple de l'adresse
  - Le numéro de ligne dans le cache
  - Le numéro d'octet dans la ligne





- 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 : $\log_2(\frac{C_s}{L_s})$ 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)
  - 0b0110010001010100
  - N° de ligne : 10
  - N° d'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

Intérêt de l'associativité

- Taille de la mémoire physique $\gg$ taille du cache
- La fonction de placement peut engendrer des conflits entre les données

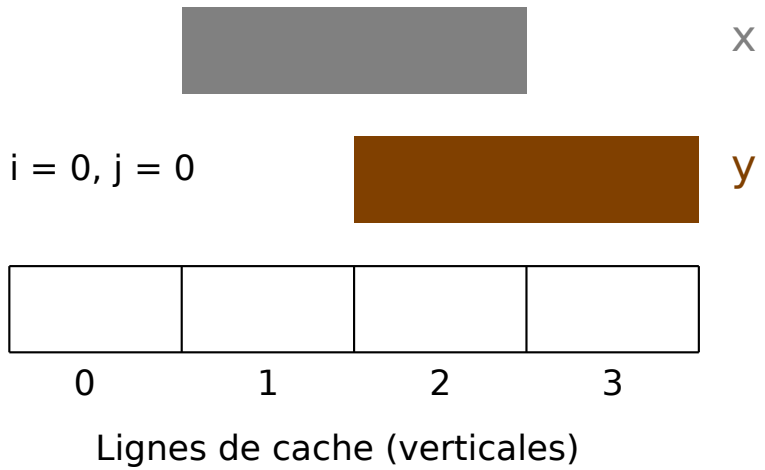
Exemple

- $C_s = 32$ octets, $L_s = 8$ octets
- x et y flottants double precision (8 octets)

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

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

- @x = 0110010001001000
- @y = 0000100000010000
- Exemple avec N = 2 :



Intérêt de l'associativité

- Taille de la mémoire physique $\gg$ taille du cache
- La fonction de placement peut engendrer des conflits entre les données

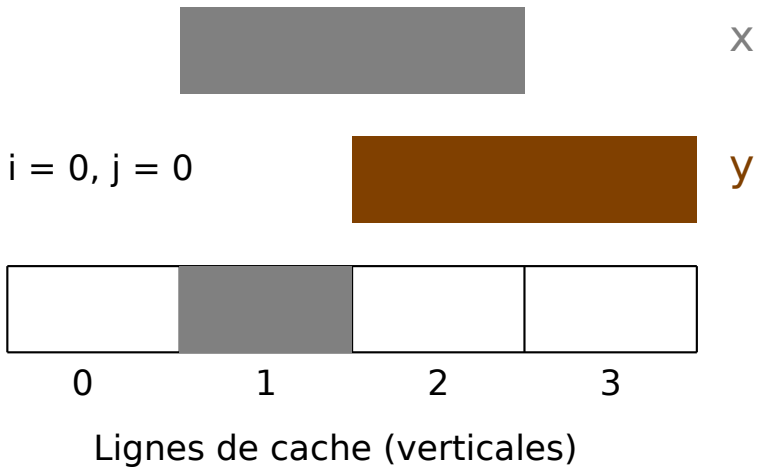
Exemple

- $C_s = 32$ octets, $L_s = 8$ octets
- x et y flottants double precision (8 octets)

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

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

- @x = 0110010001001000
- @y = 0000100000010000
- Exemple avec N = 2 :



Intérêt de l'associativité

- Taille de la mémoire physique $\gg$ taille du cache
- La fonction de placement peut engendrer des conflits entre les données

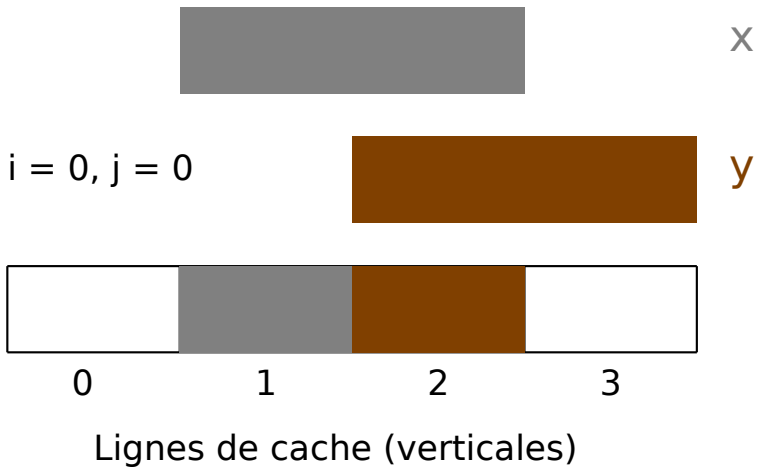
Exemple

- $C_s = 32$ octets, $L_s = 8$ octets
- x et y flottants double precision (8 octets)

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

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

- @x = 0110010001001000
- @y = 0000100000010000
- Exemple avec N = 2 :



Intérêt de l'associativité

- Taille de la mémoire physique $\gg$ taille du cache
- La fonction de placement peut engendrer des conflits entre les données

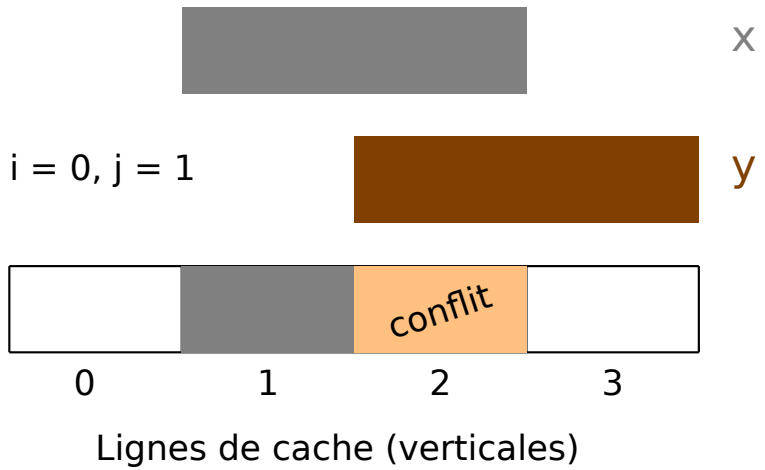
Exemple

- $C_s = 32$ octets, $L_s = 8$ octets
- x et y flottants double precision (8 octets)

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

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

- @x = 0110010001001000
- @y = 0000100000010000
- Exemple avec N = 2 :



Intérêt de l'associativité

- Taille de la mémoire physique $\gg$ taille du cache
- La fonction de placement peut engendrer des conflits entre les données

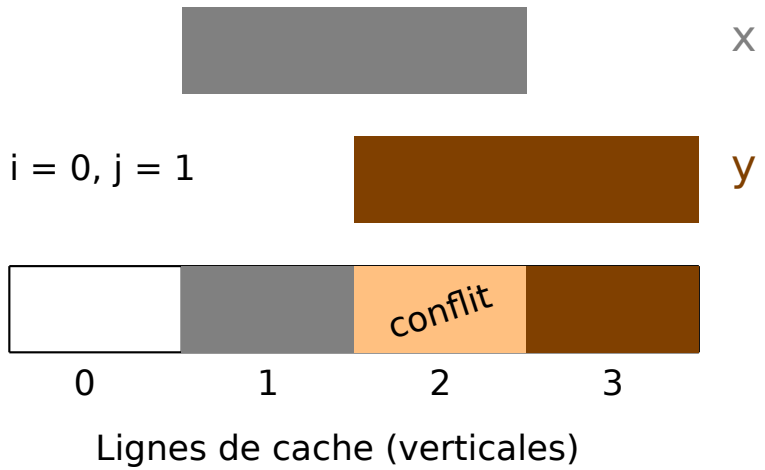
Exemple

- $C_s = 32$ octets, $L_s = 8$ octets
- x et y flottants double precision (8 octets)

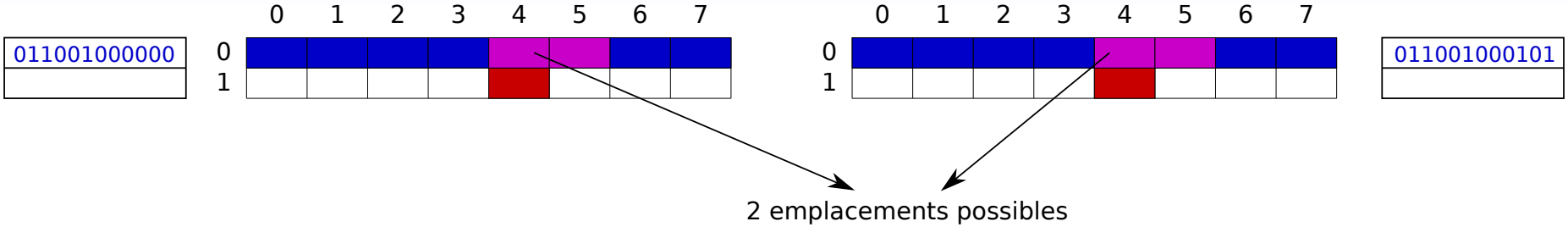
- On peut réduire les conflits en augmentant l'**associativité** des caches

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

- @x = 0110010001001000
- @y = 0000100000010000
- Exemple avec N = 2 :



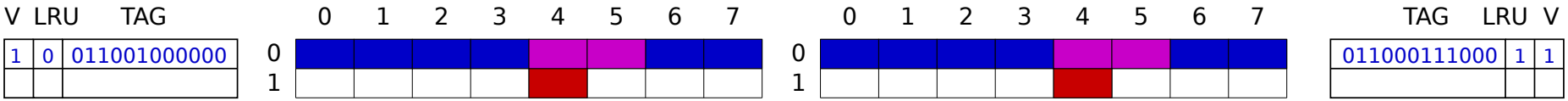
Lecture d'une donnée – Cache associatif



- Exemple :
  - $C_s = 32$ octets
  - $L_s = 8$ octets
  - A $= 2$
  - Requête de 2 octets
- Adresse demandée (16 bits) :
  - 0110010001010100
  - N° d'ensemble (index) : 0
  - N° d'octet dans la ligne (offset) : 100

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*) : le bloc remplacé est le bloc le moins récemment utilisé
  - FIFO (*First In First Out*)
  - Aléatoire
  - Pseudo-LRU : la ligne la plus récemment accédée n'est pas remplacée ; choix aléatoire entre les autres lignes



Exemple d'implémentation du pseudo-LRU : avec un bit "LRU"
On cherche d'abord une ligne invalide, puis si on n'en trouve pas, une ligne avec le bit LRU à 0.

Politiques d'écriture et d'allocation

Politique d'écriture

- **Write-Through** (cohérence au plus tôt) : toutes les écritures sont propagées en mémoire
- **Write-Back** (cohérence au plus tard) : les écritures sont locales au cache

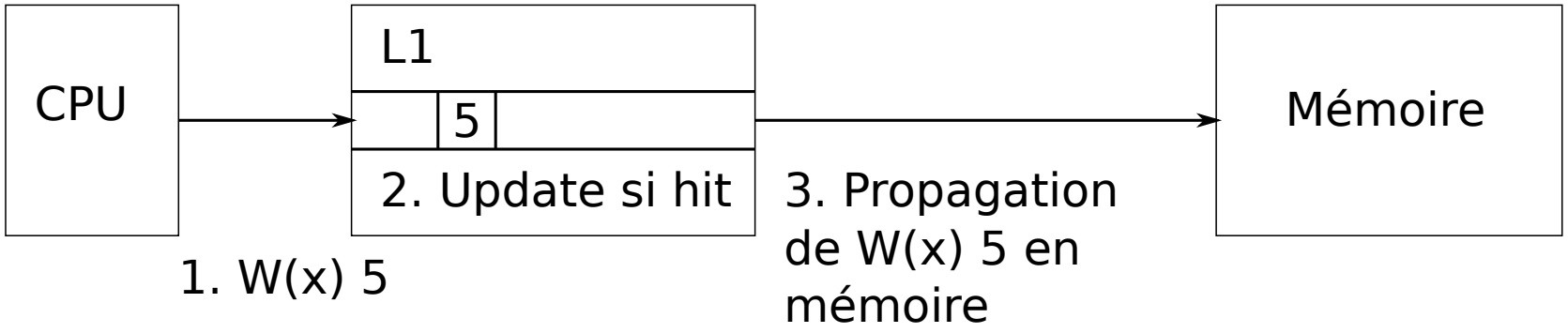
Politique d'allocation

- **Write Allocate** : la ligne est chargée lors d'un write-miss
- **Write Non-Allocate** : la ligne n'est pas chargé lors d'un write-miss

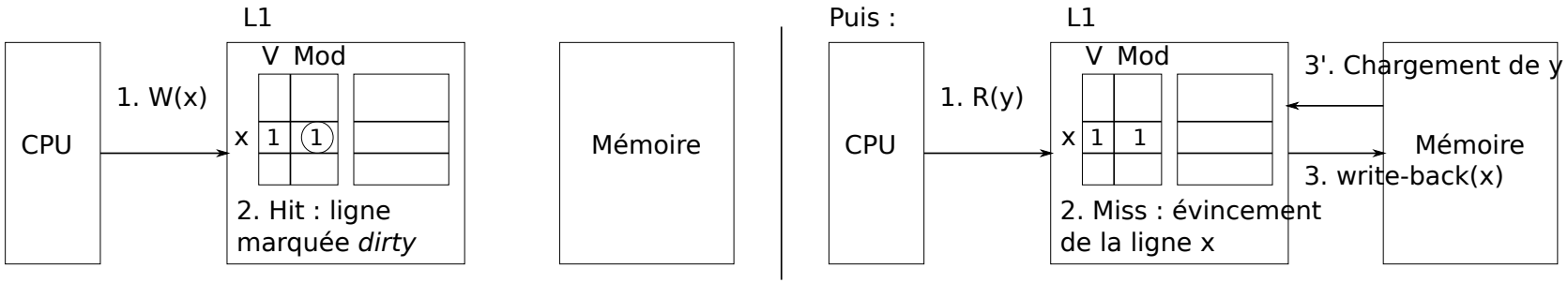
En cas de Write Miss :		Ecriture dans le cache	
		Oui	Non
Ecriture dans la mémoire	Oui	Write-Through Write Allocate	Write-Through Write Non-Allocate Write-Back Write Non-Allocate
	Non	Write-Back Write Allocate	

Politiques d'écriture

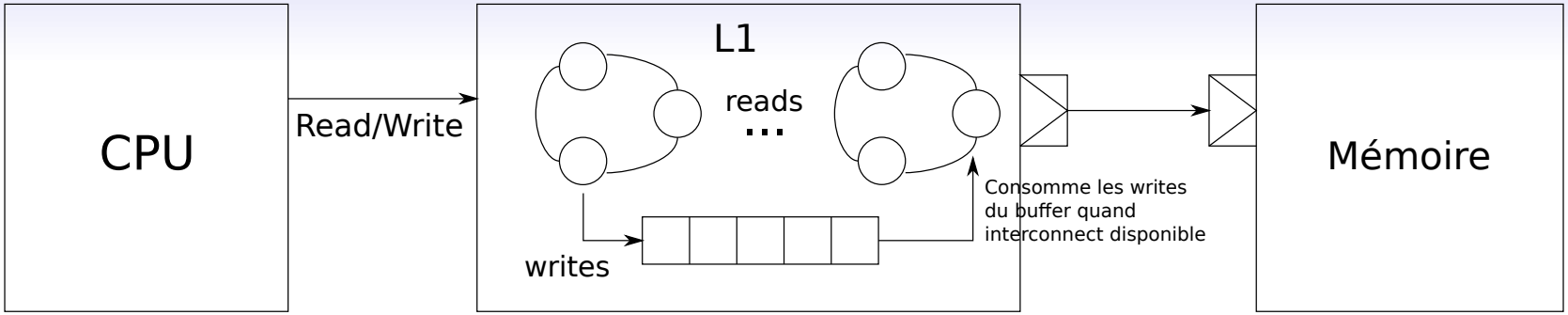
- **Write-Through**



- **Write-Back**



Write buffer pour la politique d'écriture write-through



- **Write buffer** : buffer (tampon mémoire) du cache en charge de consommer les requêtes d'écriture du processeur
- Principe de fonctionnement
  - Le processeur envoie une requête d'écriture au cache
  - Le contrôleur de cache écrit la requête dans le *write buffer*
  - Le contenu du write buffer est propagé en mémoire quand l'interconnect est disponible
- Avantages
  - Possibilité de regrouper les écritures (burst)
  - Ecritures non bloquantes en général

Performance des caches et de la hiérarchie mémoire

Temps d'exécution

- $ExecTime = InstructionCount\ (IC) \times CyclesPerInstruction\ (CPI) \times CycleTime$
- $ExecTime = IC \times (CPI_{Execution} + CPI_{MissInst} + CPI_{MissData}) \times CycleTime$
- $ExecTime = IC \times (CPI_{Execution} + MissRate_{inst} \times MissPenalty_{Inst} + \frac{MemAccess}{Inst} \times MissRate_{data} \times MissPenalty_{data}) \times CycleTime$
- $CPI_{Execution}$ inclut les instructions ALU et les accès mémoire qui font hit

Performance de la hiérarchie mémoire

- $AMAT = Average\ Memory\ Access\ Time$ (nombre de cycles par instruction)
- $AMAT_{inst} = HitTime_{inst} + MissRate_{inst} \times MissPenalty_{inst}$
- $AMAT_{data} = HitTime_{data} + MissRate_{data} \times MissPenalty_{data}$
- $AMAT = \frac{Inst}{MemAccess} \times AMAT_{inst} + \frac{DataInst}{MemAccess} \times AMAT_{data}$

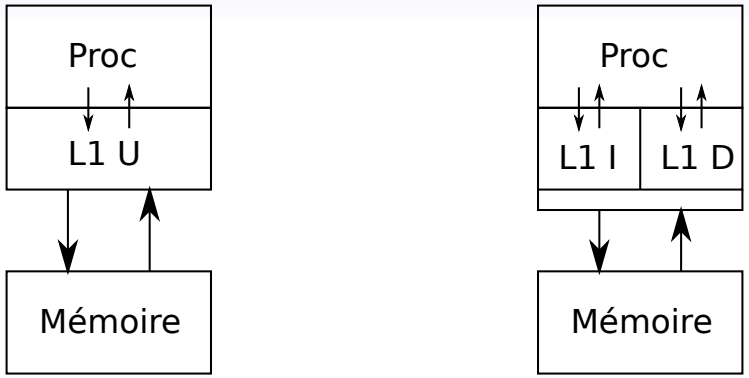
Impact sur la performance

- Hypothèses
  - CPI_{ideal} (inclut instructions ALU et les accès mémoire qui font hit) = 1.1
  - Hit Time = 1 cycle
  - Instructions : 50% arithmétique/logique, 30% load/store, 20% contrôle
  - 10% des lectures de données font miss, avec une pénalité de 50 cycles
  - 1% des instructions font miss, avec une pénalité de 50 cycles
- $CPI = CPI_{ideal}$+ Nombre moyen de cycles d'attente par instruction
  $= 1.1 + [0.30 \times 0.10 \times 50] + [1 \times 0.01 \times 50]$
  $= 1.1 + 1.5 + 0.5 = 3.1$
- 64% du temps, le processeur attend la mémoire !
- $AMAT = \frac{1}{1.3} \times [1 + 0.01 \times 50] + \frac{0.3}{1.3} \times [1 + 0.1 \times 50] = 2.54$

Impact sur la performance

- Hypothèses
  - Identiques à celles du slide précédent
  - Variation du nombre de miss de données : 100%, 50%, 10%, 1%, 0%
- $AMAT = \frac{1}{1.3} \times [1 + 0.01 \times 50] + \frac{0.3}{1.3} \times [1 + 1 \times 50] = 12.9$
- $AMAT = \frac{1}{1.3} \times [1 + 0.01 \times 50] + \frac{0.3}{1.3} \times [1 + 0.5 \times 50] = 7.15$
- $AMAT = \frac{1}{1.3} \times [1 + 0.01 \times 50] + \frac{0.3}{1.3} \times [1 + 0.1 \times 50] = 2.54$
- $AMAT = \frac{1}{1.3} \times [1 + 0.01 \times 50] + \frac{0.3}{1.3} \times [1 + 0.01 \times 50] = 1.5$
- $AMAT = \frac{1}{1.3} \times [1 + 0.01 \times 50] + \frac{0.3}{1.3} \times [1 + 0 \times 50] = 1.38$

Cache unifié vs. caches séparés



- Exemple
  - 16KB Inst. & 16KB Données – Taux de miss inst. : 0.64%, taux de miss données : 6.47%
  - 32KB unifié – Taux de miss global : 1.99%
- Comparaison
  - 30% des instructions sont des accès mémoire
  - Temps d'accès : 1 cycle – Pénalité d'échec : 50 cycle
  - Un accès données sur le cache unifié $\Rightarrow$ 1 cycle de gel (un seul port de lecture)
- $AMAT_{Harvard} = \frac{1}{1.3} \times [1 + 0.64\% \times 50] + \frac{0.3}{1.3} \times [1 + 6.47\% \times 50] = 2.0$
- $AMAT_{Unified} = \frac{1}{1.3} \times [1 + 1.99\% \times 50] + \frac{0.3}{1.3} \times [1 + 1 + 1.99\% \times 50] = 2.23$

Améliorer les performances des caches

- Temps d'accès moyen à la mémoire :

  AMAT = Temps de succès + Taux d'échecs × Pénalité d'échec
- 3 moyens pour améliorer la performance
  - Réduire le taux d'échecs
  - Réduire la pénalité d'un échec
  - Réduire le temps d'accès au cache (en cas de 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 blocs
- Augmenter l'associativité
- Le *Victim Cache*
- Les cache Pseudo-associatifs
- Les caches *Skew associative*
- Le préchargement matériel
- Des optimisations à la compilation

Les types d'échec : les 3Cs

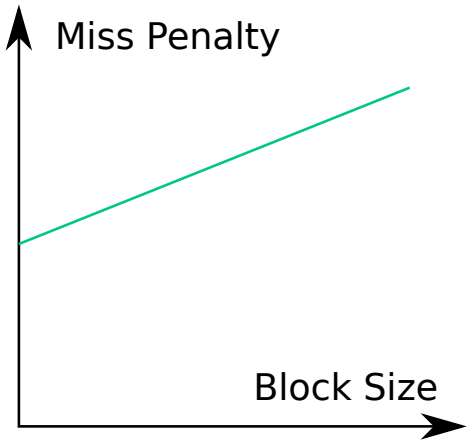
- **Compulsory misses** (cold/startup misses) : le premier accès à un bloc absent du cache. Échec même en cas de cache infini
- **Capacity misses** (ensemble de données trop grand) : échec sur une même donnée, même en cas de cache complètement associatif
- **Conflict misses** (interference misses) : échec dû à l'utilisation d'un même ensemble (set), le degré d'associativité est insuffisant
- **Un 4^{eme} C : Cohérence** : échec causé par la gestion de la cohérence de cache (cf. cours futur)

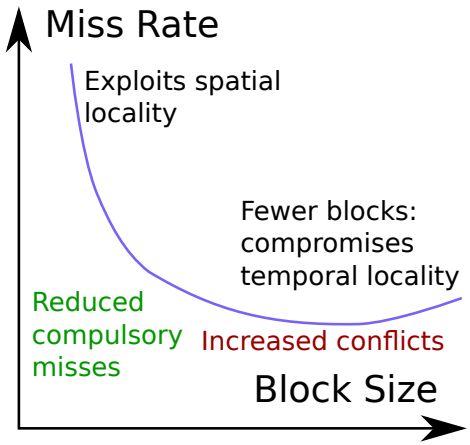
Augmenter la taille du cache

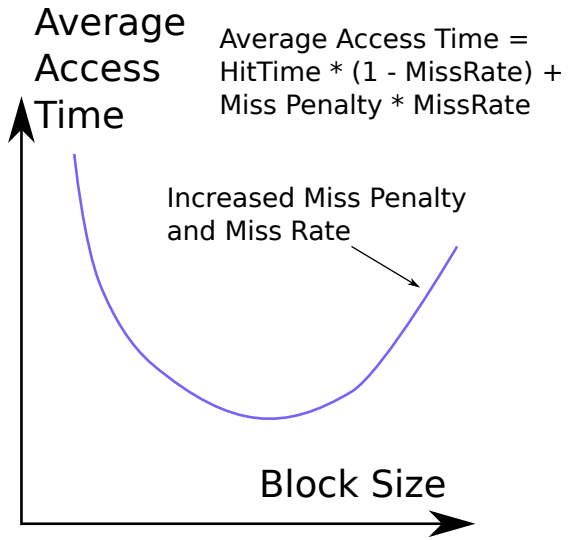
- Pas d'effet sur les *compulsory misses* (échec 1^{er} accès)
- Réduit les *capacity misses* (plus de capacité)
- Réduit les *conflict misses* (en général)
  - Peu de blocs en concurrence en moyenne pour un ensemble (beaucoup d'ensembles)
  - Probabilité plus faible que le nombre de blocs actifs d'un ensemble soit supérieur à la taille de l'ensemble
- Mais augmentation du temps d'accès (et du cout !)

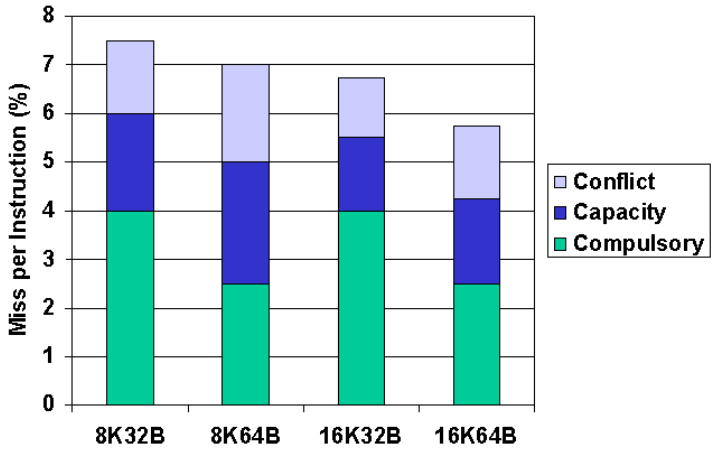
Augmenter la taille du bloc

- En général, un bloc large exploite la localité spatiale (réduit les *compulsory misses*), mais
  - Augmente le temps de transfert
  - Augmente les miss de conflit, car il y a moins de blocs pour une taille de cache donnée



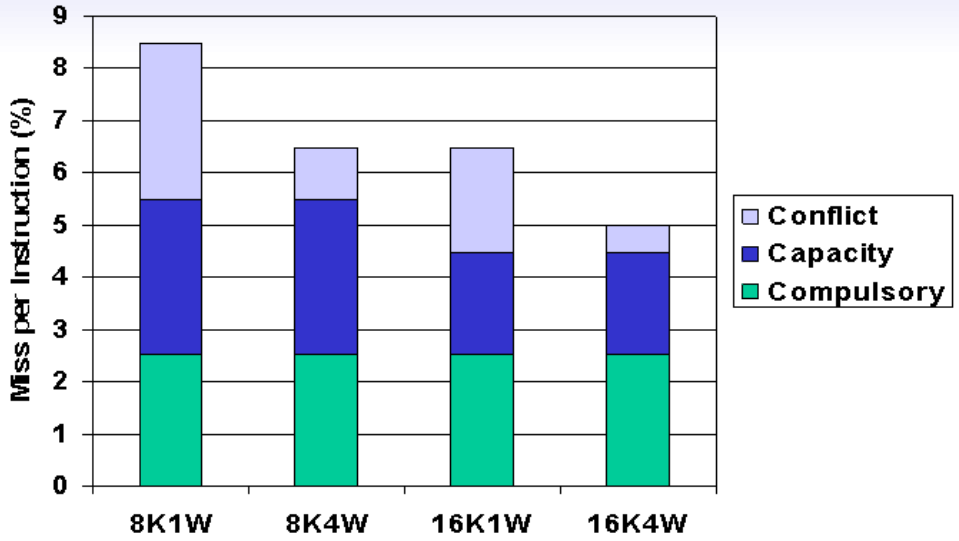


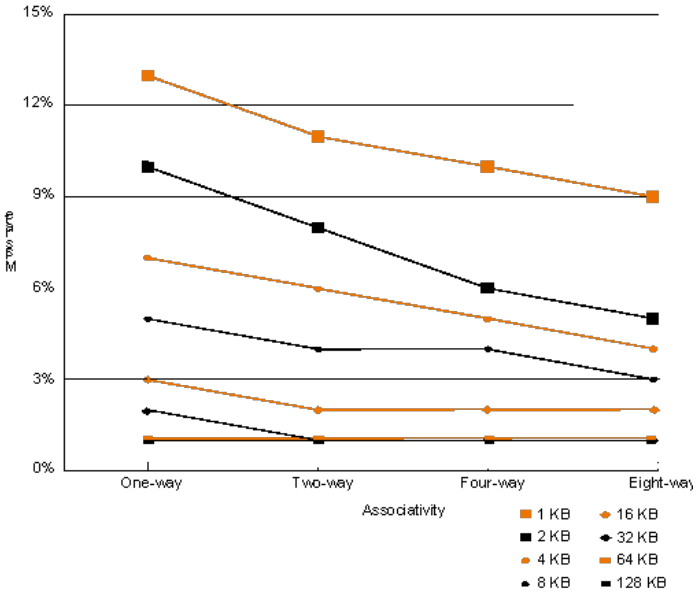


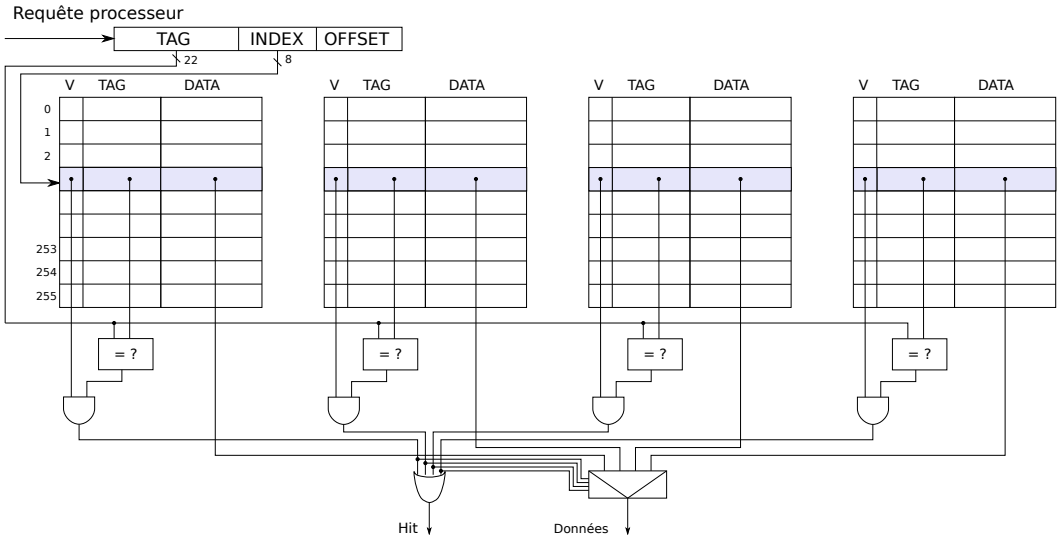


Augmenter l'associativité

- Avantage :
  - Réduction des *conflict misses*
- Désavantages :
  - Augmentation du temps d'accès
  - Nécessite plus de matériel (comparateurs, muxes, tags)
  - Diminution des bénéfices avec le passage à 4 ou 8 voies

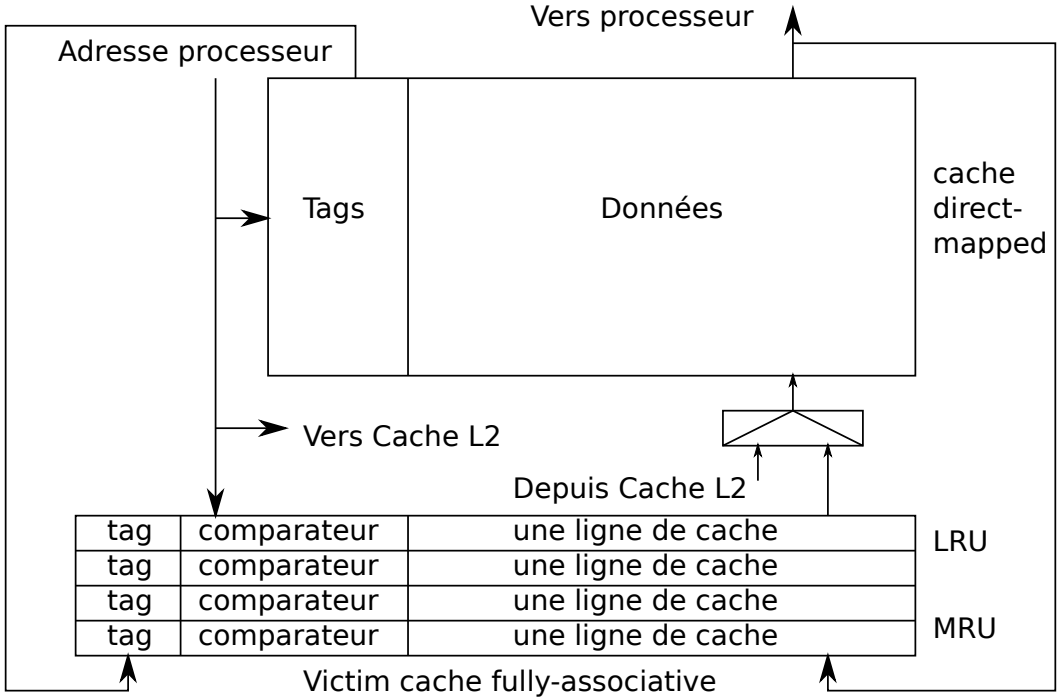






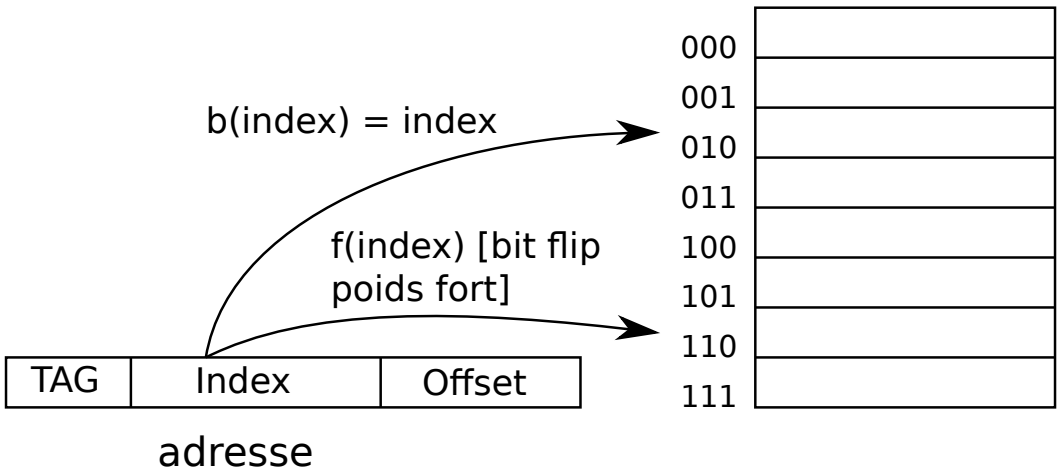
Le Victim Cache

- En général, peu de lignes sont en conflit à un moment donné
- Pas nécessaire d'avoir un cache entier associatif
- Idée : avoir un petit cache derrière le cache L1, entièrement associatif
- Ne rallonge pas le chemin critique, contrairement à un cache associatif
- $\Rightarrow$ Réduit les miss de conflit



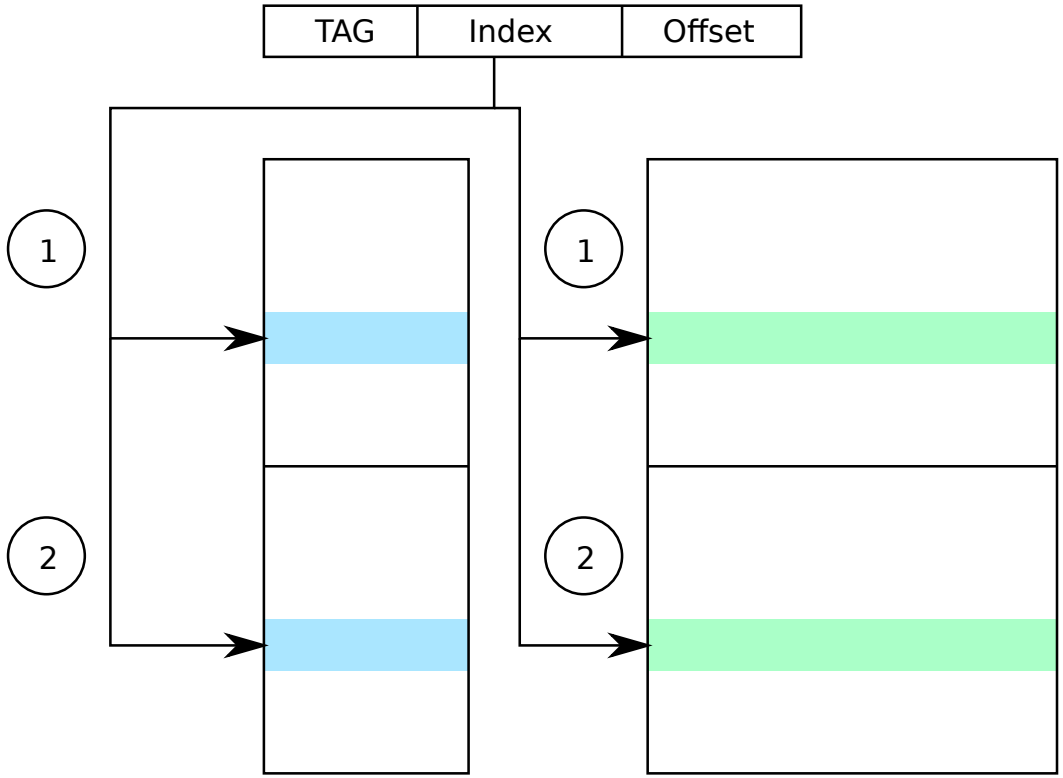
Les caches column-associative

- Test du cache avec index calculé par b (typiquement identité)
  - Si hit, accès en un cycle, temps de hit = temps de hit *direct-mapped*
- Si echec, test avec f (rehash)
  - Si hit, accès en 2 cycles
- Si nouvel échec, chargement à un des deux emplacements (b ou f) + évincement de l'ancienne ligne, et swap si chargement à f (pour avoir un hit en 1 cycle au prochain accès)
- Un bit de plus par ligne dans le TAG car un bit de moins dans l'index (comme pour les caches *2-way associative*)
- Besoin d'un bit supplémentaire pour savoir si la ligne contient un *rehash* : lignes évincées en priorité
- $\Rightarrow$ Réduction des miss de conflit



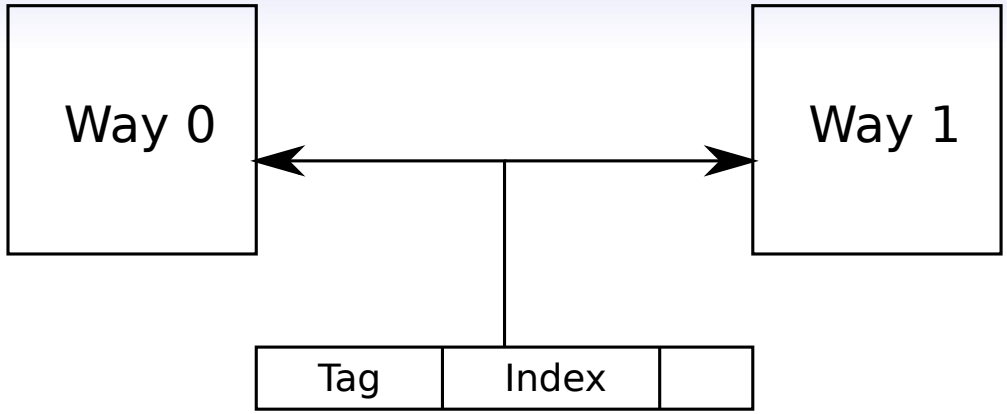
Les caches pseudo-associatifs

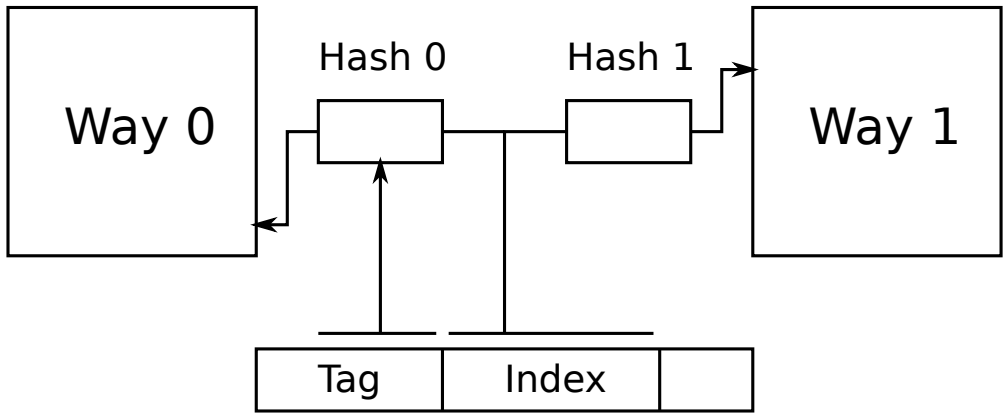
- Principe similaire aux caches column-associative : cache virtuellement découpé en 2 parties
- Nombre d'échec similaire à un cache 2-way et temps d'accès à un cache *direct-mapped* en cas de hit
- En cas d'échec à la première partie, accès à la deuxième + swap
- $\Rightarrow$ Réduction des miss de conflit
- Utilisé dans le cache L2 du Mips R1000



Les caches skew-associative

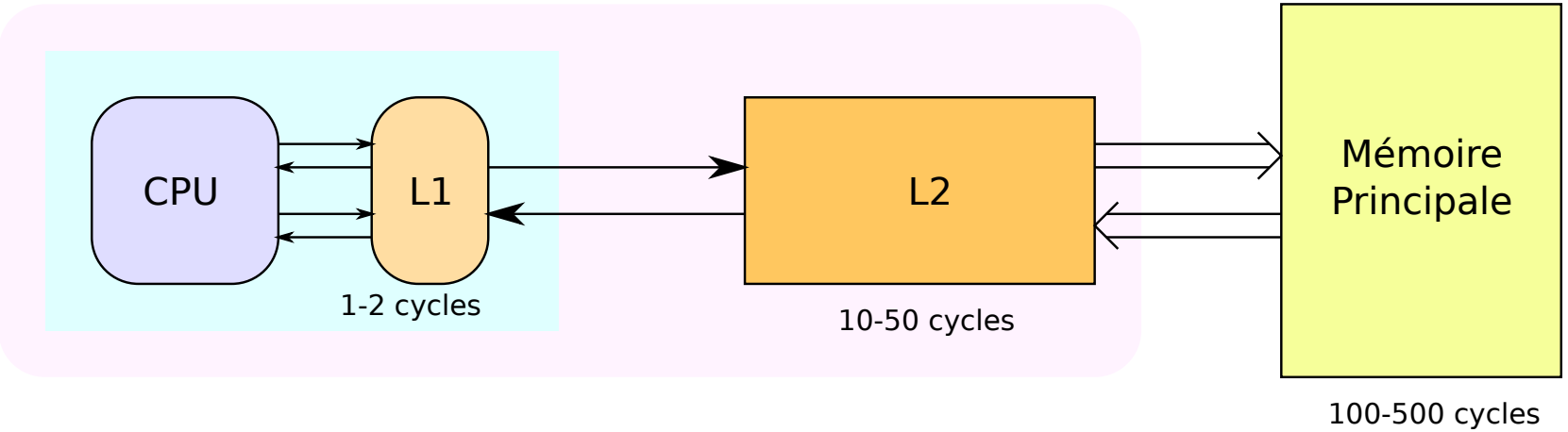
- Un miss de conflit se produit quand $n + 1$ blocs ont le même index dans un cache à n voies
- Pour réduire ces miss de conflit, utiliser des indices différents entre les voies du cache
- Exemple : *xor* entre les bits de l'index et du tag



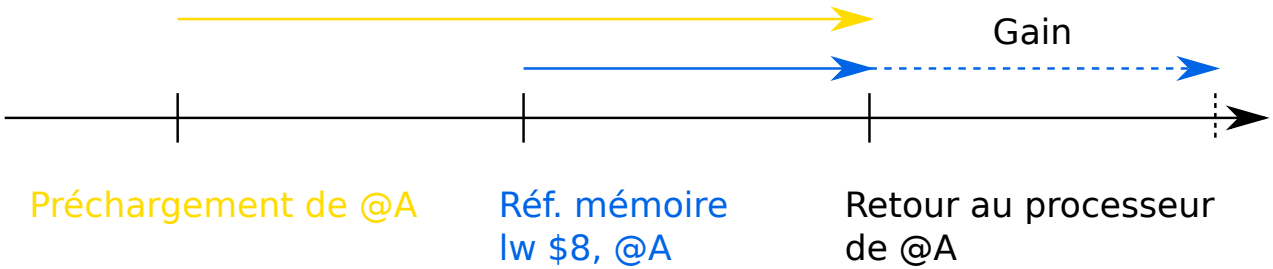


Le préchargement matériel

- **Précharger** : Anticiper la demande explicite de la donnée par le processeur
- Envoyer un requête de lecture mémoire avant la référence effective du processeur pour masquer la **latence mémoire**
- ⇒ Pouvoir **prédire** quelles données le processeur va utiliser




```
...
addu $8, $9, $10
or $10, $9, $11
lw $8, @A
lw $9, @B
...
```



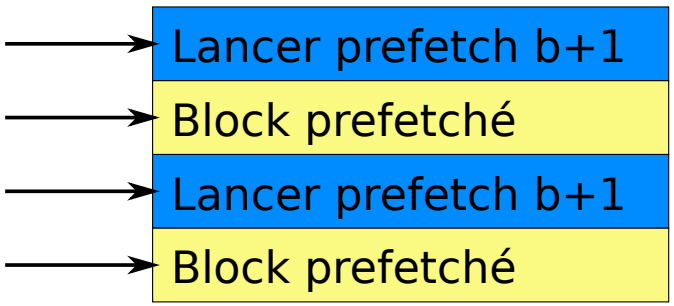
Le préchargement : prédire et stocker

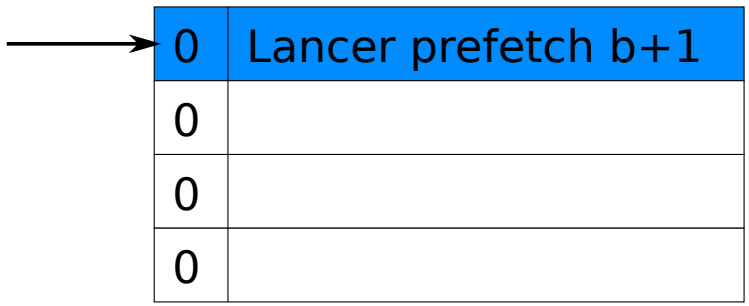
La technique de préchargement mise en oeuvre doit répondre à 3 questions :

- **What** : Quelles données précharger ?
  - Sur succès ? Sur échec ?
  - Si les données préchargées ne sont pas utilisées, remplacement de données utiles (pollution du cache), consommation inutile de la bande passante du réseau
- **When** : Quand précharger ces données ?
  - Trop tôt : pollution du cache ou remplacement de la ligne préchargée avant son utilisation
  - Trop tard : ne masque pas la latence mémoire
- **Where** : Où stocker les données préchargées ?
  - Dans le cache ou dans un buffer spécifique (*prefetch buffer* ou buffer de préchargement) ?
  - Niveau de cache où est appliquée la prédiction : L1 ou L2

Préchargement séquentiel : sans buffer de préchargement

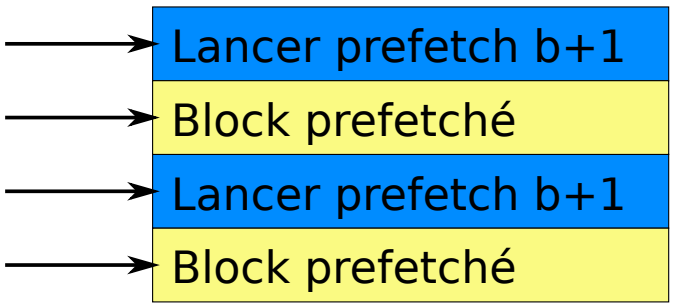
- **Préchargement séquentiel** : tirer parti de la localité spatiale ($\Rightarrow$ plutôt appliqué aux instructions)
- Initier le préchargement du bloc $b+1$ quand le bloc b est accédé
- 2 approches
  - Prefetch sur un miss : à chaque fois qu'un accès au bloc b fait miss
  - Prefetch marqué (*tagged prefetch*) : utilise un bit en plus par ligne pour faire un préchargement lors du premier accès (hit) à un bloc préchargé
- Exemple avec des accès séquentiels :

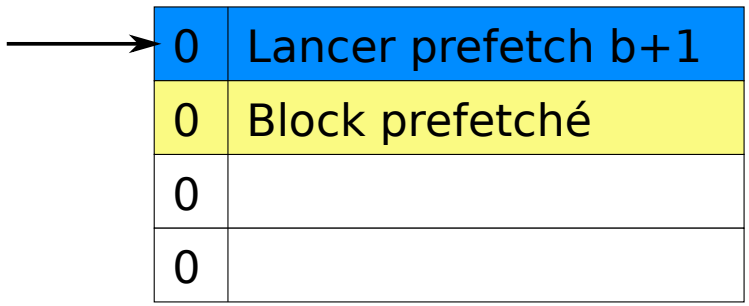




Préchargement séquentiel : sans buffer de préchargement

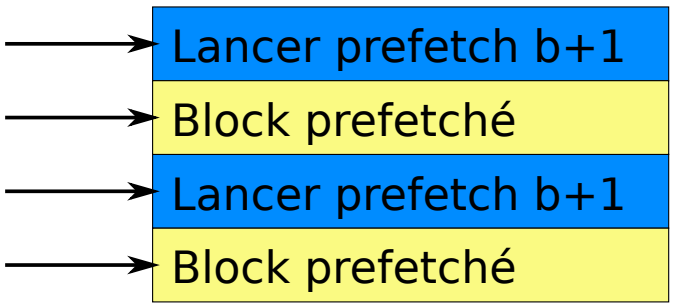
- **Préchargement séquentiel** : tirer parti de la localité spatiale ($\Rightarrow$ plutôt appliqué aux instructions)
- Initier le préchargement du bloc $b+1$ quand le bloc b est accédé
- 2 approches
  - Prefetch sur un miss : à chaque fois qu'un accès au bloc b fait miss
  - Prefetch marqué (*tagged prefetch*) : utilise un bit en plus par ligne pour faire un préchargement lors du premier accès (hit) à un bloc préchargé
- Exemple avec des accès séquentiels :

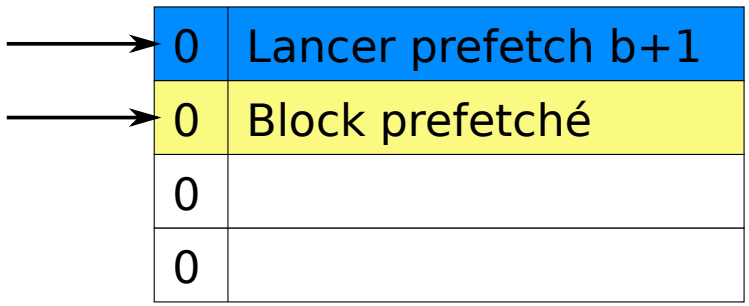




Préchargement séquentiel : sans buffer de préchargement

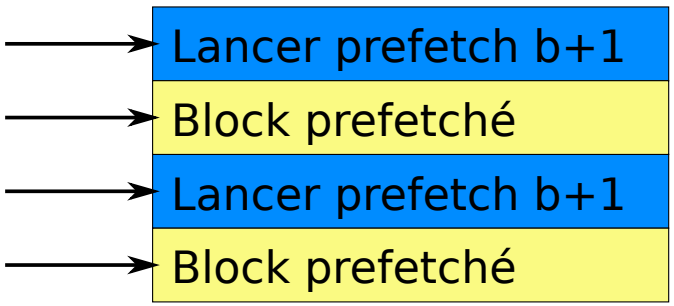
- **Préchargement séquentiel** : tirer parti de la localité spatiale ($\Rightarrow$ plutôt appliqué aux instructions)
- Initier le préchargement du bloc $b+1$ quand le bloc b est accédé
- 2 approches
  - Prefetch sur un miss : à chaque fois qu'un accès au bloc b fait miss
  - Prefetch marqué (*tagged prefetch*) : utilise un bit en plus par ligne pour faire un préchargement lors du premier accès (hit) à un bloc préchargé
- Exemple avec des accès séquentiels :

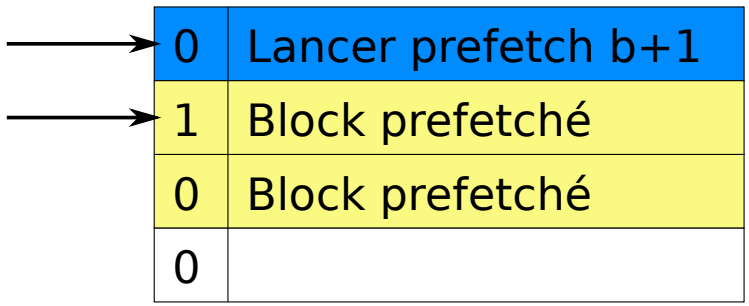




Préchargement séquentiel : sans buffer de préchargement

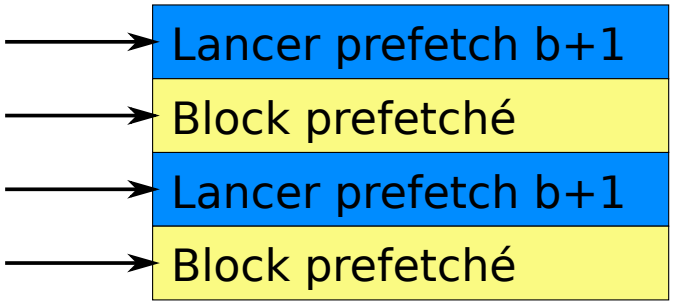
- **Préchargement séquentiel** : tirer parti de la localité spatiale ($\Rightarrow$ plutôt appliqué aux instructions)
- Initier le préchargement du bloc $b+1$ quand le bloc b est accédé
- 2 approches
  - Prefetch sur un miss : à chaque fois qu'un accès au bloc b fait miss
  - Prefetch marqué (*tagged prefetch*) : utilise un bit en plus par ligne pour faire un préchargement lors du premier accès (hit) à un bloc préchargé
- Exemple avec des accès séquentiels :

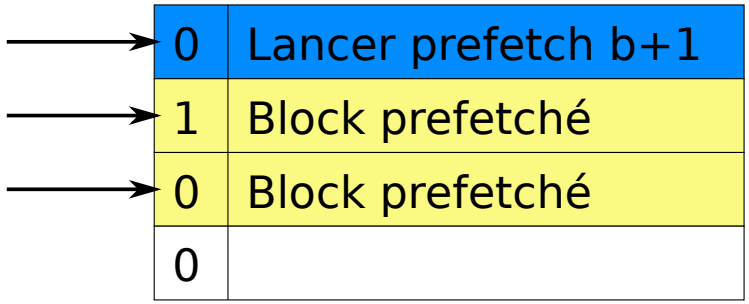




Préchargement séquentiel : sans buffer de préchargement

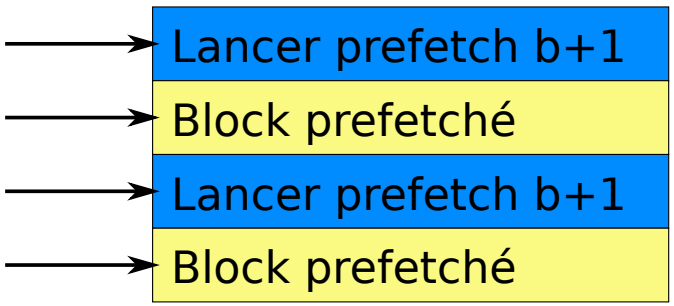
- **Préchargement séquentiel** : tirer parti de la localité spatiale ($\Rightarrow$ plutôt appliqué aux instructions)
- Initier le préchargement du bloc $b+1$ quand le bloc b est accédé
- 2 approches
  - Prefetch sur un miss : à chaque fois qu'un accès au bloc b fait miss
  - Prefetch marqué (*tagged prefetch*) : utilise un bit en plus par ligne pour faire un préchargement lors du premier accès (hit) à un bloc préchargé
- Exemple avec des accès séquentiels :

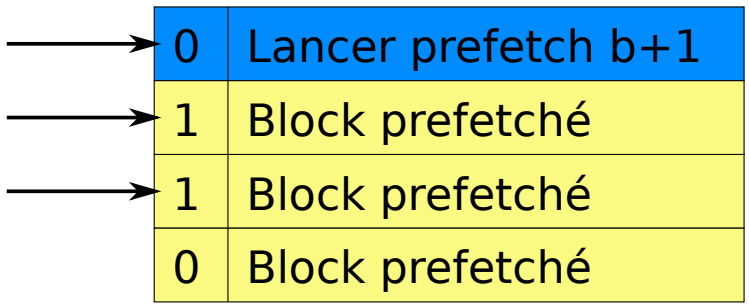




Préchargement séquentiel : sans buffer de préchargement

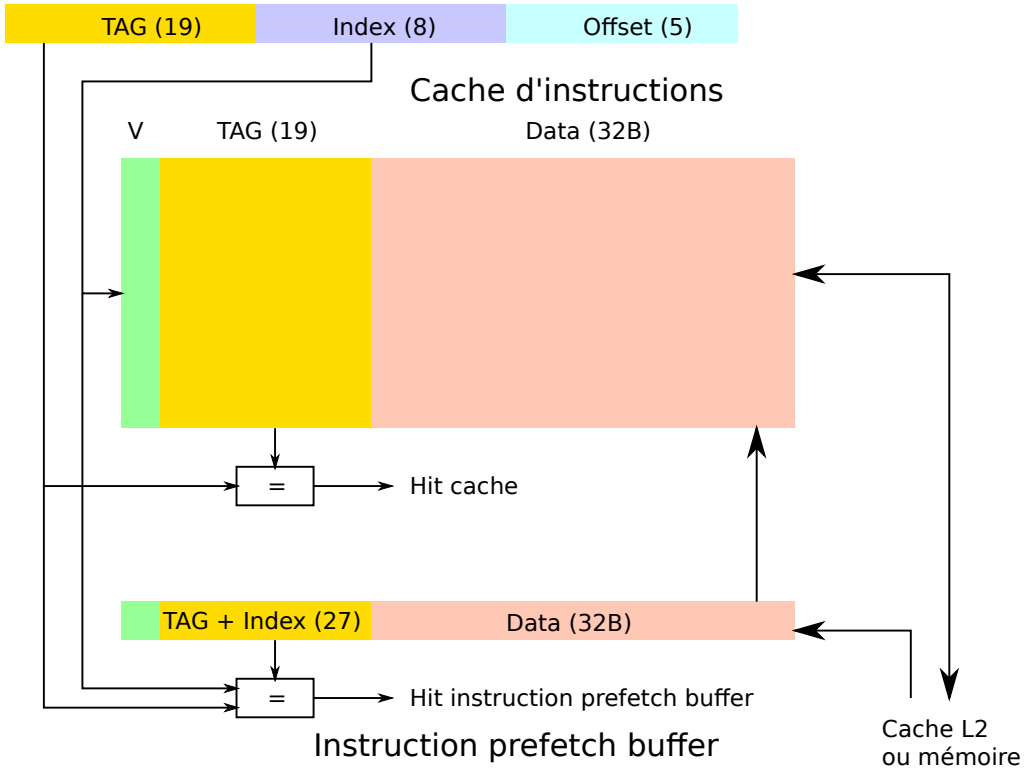
- **Préchargement séquentiel** : tirer parti de la localité spatiale ($\Rightarrow$ plutôt appliqué aux instructions)
- Initier le préchargement du bloc $b+1$ quand le bloc b est accédé
- 2 approches
  - Prefetch sur un miss : à chaque fois qu'un accès au bloc b fait miss
  - Prefetch marqué (*tagged prefetch*) : utilise un bit en plus par ligne pour faire un préchargement lors du premier accès (hit) à un bloc préchargé
- Exemple avec des accès séquentiels :





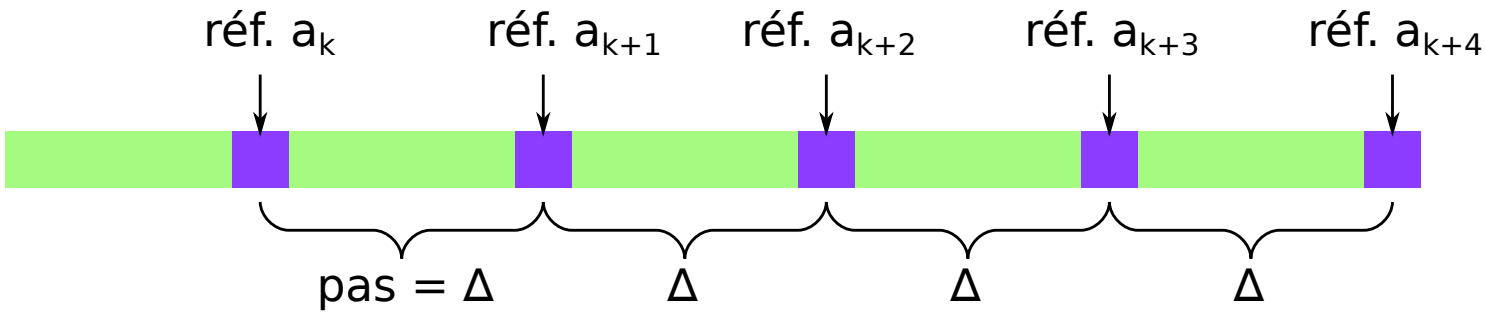
Préchargement séquentiel : avec un buffer de préchargement

- Présence d'un **buffer de préchargement** accédé en parallèle du cache à chaque requête processeur
- Lors d'un hit sur le cache, pas de modification du buffer de préchargement
- Lors d'un échec sur le cache et sur le buffer de préchargement, la ligne requise est retournée au cache instructions et la ligne suivante est préchargée dans le buffer
- Lors d'un échec sur le cache et d'un hit sur le buffer de préchargement, la ligne qu'il contient est ramenée en cache, et la ligne suivante est préchargée
- Pas besoin de bit en plus par ligne
- Plus sophistiqué : la prédiction de branchement dirige le préchargement



Préchargement avec pas

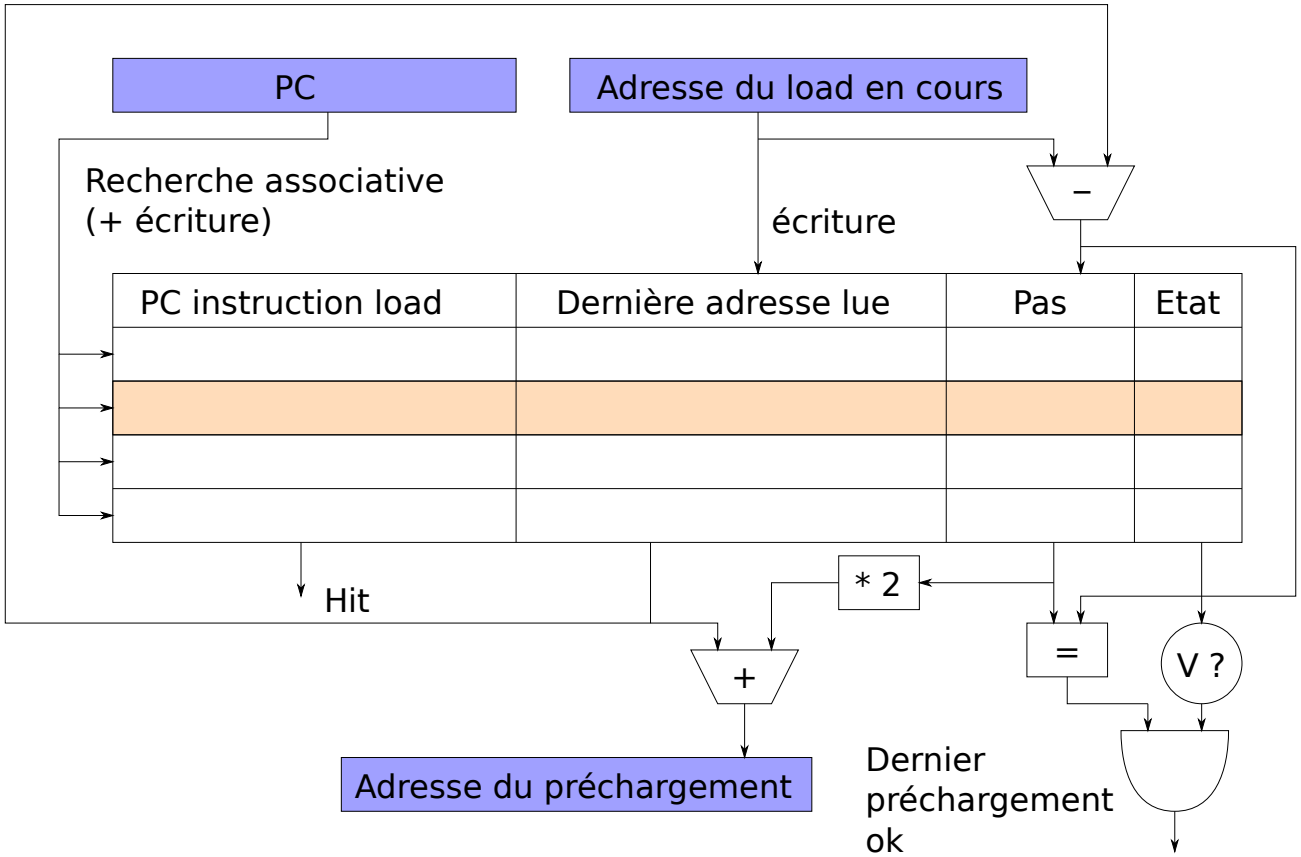
- Emploi de matériel spécifique pour surveiller le motif (*pattern*) des adresses émises par le processeur
- Détecter les accès mémoire avec un pas constant : typiquement le cas lors d'accès à des tableaux depuis des boucles
- Comparaison des adresses référencées avec les adresses prédites



- Idée : soit une instruction *inst* qui effectue un accès à l'adresse a_k (k indice de boucle)
- Le préchargement pour cette instruction sera activé si $a_1 - a_0 = \Delta \neq 0$
  - Suppose que Δ est le pas d'une série d'accès à un tableau
- La première adresse préchargée est $A_2 = a_1 + \Delta$, puis $A_k = A_{k-1} + \Delta$
- Le préchargement continue jusqu'à ce que $A_k \neq a_k$

Préchargement avec pas : matériel requis

- Table contenant les informations pour les motifs d'accès en cours et les préchargements associés
- Chaque entrée contient :
  - L'adresse de l'instruction d'accès (PC du `load`)
  - L'adresse de la dernière donnée accédée par cette instruction
  - La valeur du pas
  - L'état de l'entrée : invalide, initialisé, valide (par exemple)



Préchargement avec pas : exemple

```
int a[128][128];
int b[128][128];
int c[128][128];
...
for (int i = 0; i < 128; i++) {
  for (int j = 0; j < 128; j++) {
    c[i][j] = 0;
    for (int k = 0; k < 128; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

- Exemple de table après l'instruction d'affectation pour i = 0, j = 0, k = 2

PC instruction `load`	Dernière adresse lue	Pas	État
`@lw c[i][j]`	0x10030000	?	Initialisé
`@lw a[i][k]`	0x10010008	4	Valide
`@lw b[k][j]`	0x10020400	0x200 = 512	Valide

Optimisations à la compilation

- Objectif : augmenter la localité spatiale et temporelle des codes
- Suivant les cas, peut être fait à la main, par le compilateur, ou les deux
- Exemples de techniques :
  - Regroupement de tableaux
  - Permutation de boucles
  - Fusion de boucles
  - *Blocking*
  - *Padding*
  - Préchargement logiciel

Regroupement de tableaux

```c
/* Code original : deux tableaux disjoints en mémoire */
int pos_x[SIZE];
int pos_y[SIZE];

/* Code modifié : un tableau de structures */
// pos_x[i] (pos[i].x) et pos_y[i] (pos[i].y) sont contigüs en mémoire
typedef struct _pos_s {
    int x;
    int y;
} pos_s;
pos_s pos[SIZE];
```

- Réduction des conflits entre `pos_x` et `pos_y` (surtout si `SIZE` est un multiple de la taille du cache)
- 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

```
int tab[5000][100];

/* Code original */
for (k = 0; k < 100; k++) {
   for (j = 0; j < 100; j++) {
      for (i = 0; i < 5000; i++) {
         tab[i][j] = 2 * tab[i][j];
      }
   }
}
```

```
int tab[5000][100];

/* Code modifié */
for (k = 0; k < 100; k++) {
   for (i = 0; i < 5000; i++) {
      for (j = 0; j < 100; j++) {
         tab[i][j] = 2 * tab[i][j];
      }
   }
}
```

- Accès séquentiel vs. accès avec un pas de 100
- Augmentation de la localité spatiale

Fusion de boucles

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

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

```
/* Code modifié */
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    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 et c contre 1 miss par accès
- Augmentation de la localité temporelle

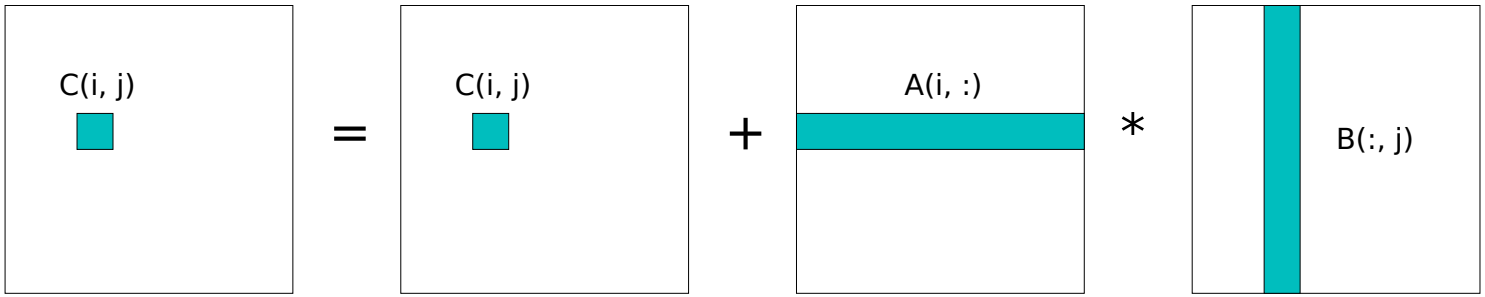
Multiplication de matrices

- Calcul de $C = A \times B$

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < N; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

Réutilisation sur j

Réutilisation sur i

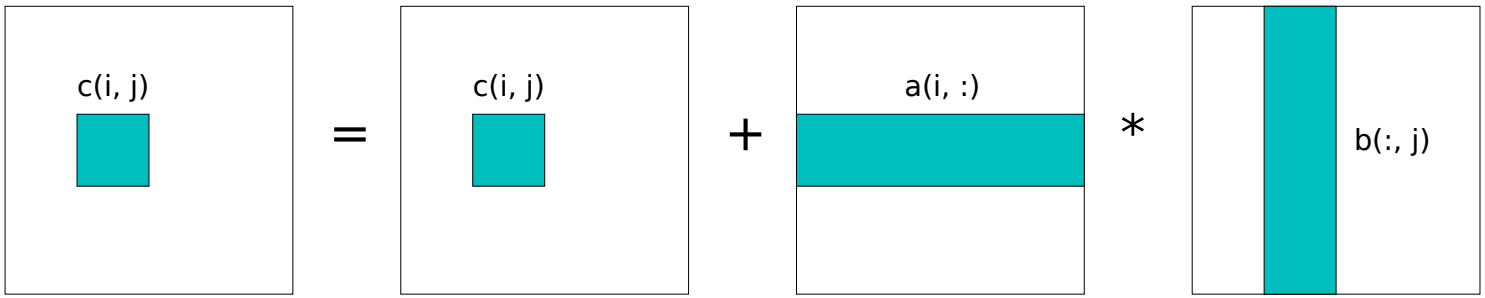


- Si le cache ne peut pas contenir une ligne de A entièrement, il y aura un miss sur `A[i][k]` à chaque fois que l'on passe de C(i, j) à C(i, j + 1)
- Si le cache ne peut pas contenir B entièrement, il y aura un miss sur `B[k][j]` à chaque accès (encore pire !)

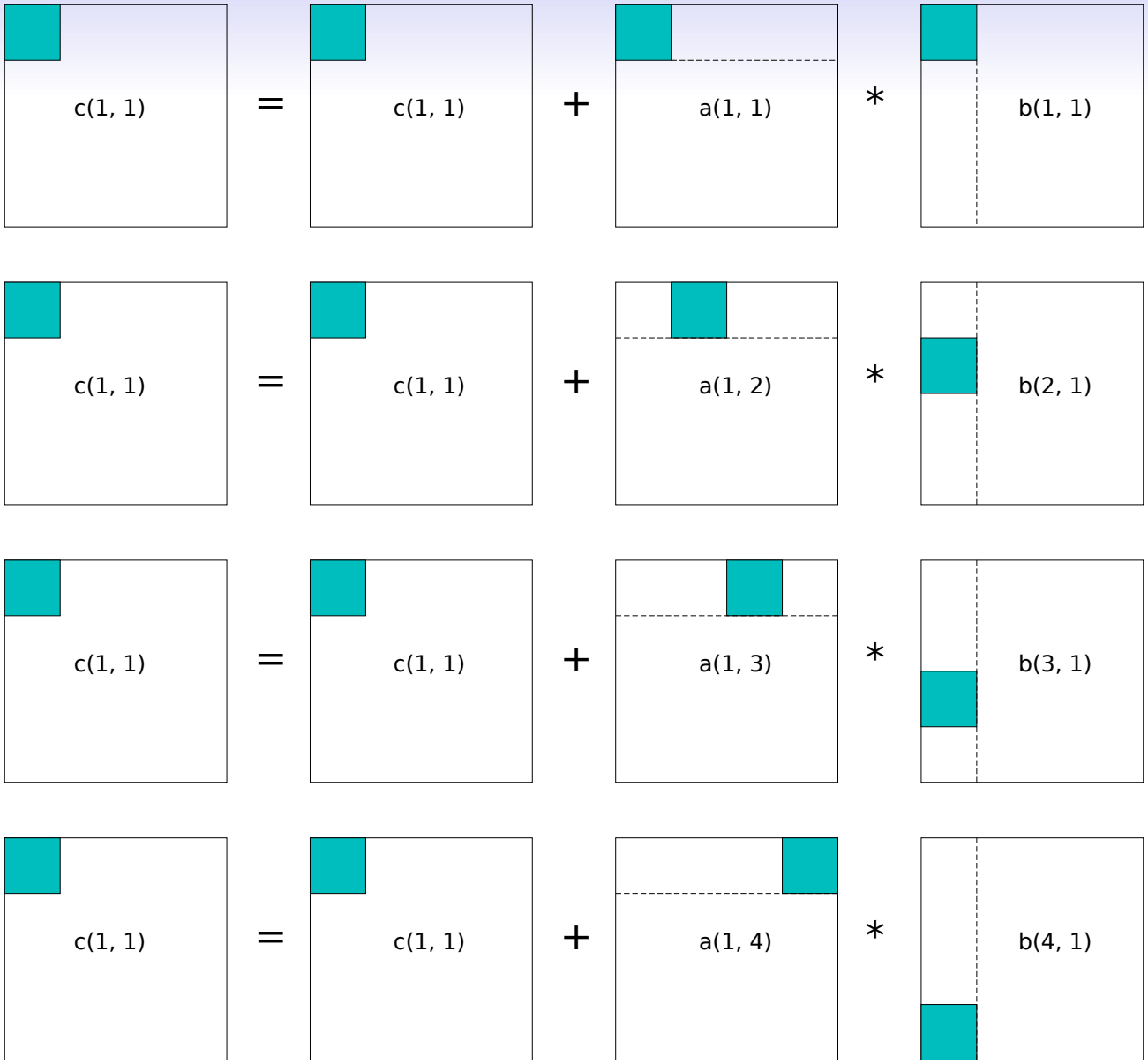
Blocking

- Idée : 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 $n \times n$ où $n = \frac{N}{\#blocs}$

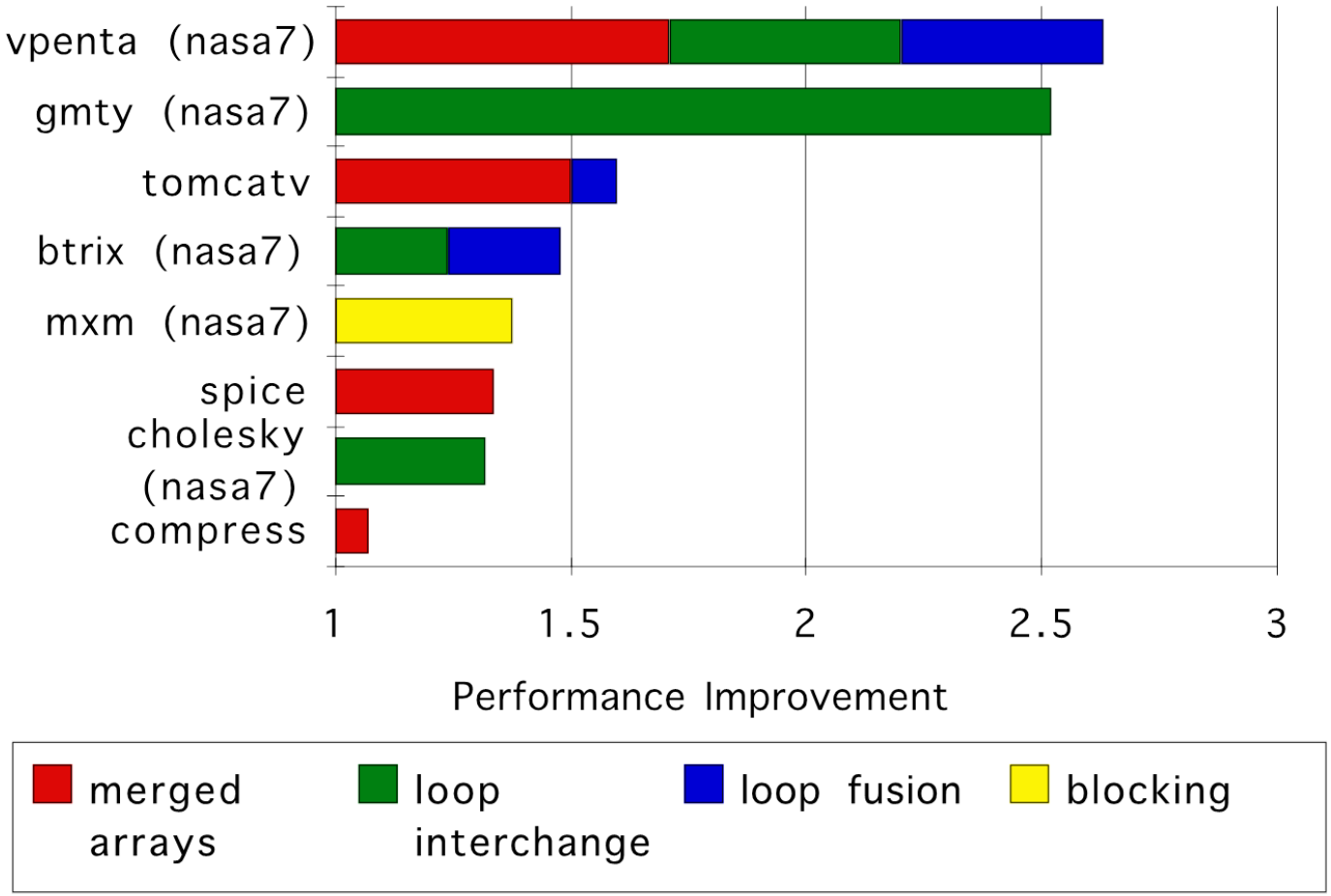
```
for (int i = 0; i < N / n; i++) {
    for (int j = 0; j < N / n; j++) {
        {read block c(i, j) into fast memory}
        c(i, j) = 0;
        for (int k = 0; k < N / n; k++) {
            {read block a(i, k) into fast memory}
            {read block b(k, j) into fast memory}
            c(i, j) = c(i, j) + a(i, k) * b(k, j);
        }
        {write block c(i, j) back to slow memory}
    }
}
```



Multiplication de matrices par blocs



Performances des optimisations



Padding de tableaux

- But : éviter les miss de conflit
- Hypothèses :
  - Taille du cache : 8 Ko
  - Taille des double : 8 octets
  - Taille du bloc : 32 octets

```
/* Code original */
int CS = 1024;
double A[CS], B[CS];
for (int i = 0; i < CS; i++) {
   A[i] = A[i] + b * B[i];
}
```

```
/* Code modifié */
int CS = 1024;
double A[CS];
// padding de 4 double
double pad_buffer[4];
double B[CS];
for (int i = 0; i < CS; i++) {
   A[i] = A[i] + b * B[i];
}
```

- 2 miss toutes les 4 opérations vs. 2 miss par opération

Préchargement logiciel

- Les jeux d'instructions peuvent comporter des instructions spéciales permettant de lancer le préchargement
- Instructions écrites à la main ou générées par le compilateur à la suite de l'analyse des dépendances et des accès

```
/* Code original */
int A[N], B[N];
for (int i = 0; i < N; i++) {
   A[i] = A[i] + b * B[i];
}
```

```
/* Code modifié */
int A[N], B[N];
fetch(&A[0]);
fetch(&B[0]);
for (int i = 0; i < N - 1; i++) {
   fetch(&A[i + 1]);
   fetch(&B[i + 1]);
   A[i] = A[i] + b * B[i];
}
```

- Problème : une ligne de cache contient plus d'un élément (par exemple, 4) $\Rightarrow$ certains préchargements inutiles (ex : `&A[1], &A[2]`)

Préchargement logiciel (suite)

- Solution : dérouler la boucle sur une ligne de cache

```
/* Code avec déroulage et prefetch */
int A[N], B[N];
fetch(&A[0]);
fetch(&B[0]);
for (i = 0; i < N - 4; i += 4) {
    fetch(&A[i + 4]);
    fetch(&B[i + 4]);
    A[i] = A[i] + b * B[i];
    A[i + 1] = A[i + 1] + b * B[i + 1];
    A[i + 2] = A[i + 2] + b * B[i + 2];
    A[i + 3] = A[i + 3] + b * B[i + 3];
}
for (i = N - 4; i < N; i++) {
    A[i] = A[i] + b * B[i];
}
```

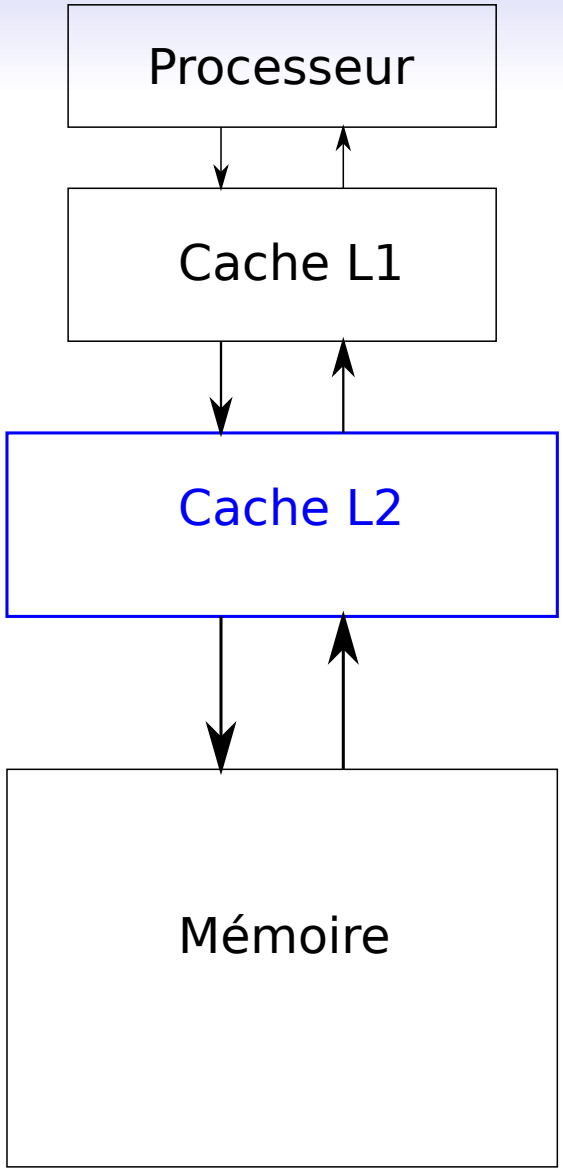
- Fait l'hypothèse que précharger les éléments de l'itération suivante est suffisant pour masquer la latence
- Peut ne pas être suffisant $\Rightarrow$ Adapter le code pour précharger les éléments accédés dans k itérations

Quelques techniques pour réduire la pénalité d'échec

- Un second niveau de cache
- Redémarrage précoce et mot demandé en premier
- Les caches non-bloquants
- Accélérer la traduction d'adresse

Caches de second niveau

- Pénalité en cas d'échec dans le L1 est élevé car la latence d'accès mémoire est élevée
- Comment masquer cette latence ? En ajoutant une mémoire cache entre le cache et la mémoire : cache de second niveau
- Second niveau de cache plus grand $\Rightarrow$ meilleur taux de succès et temps de succès $\ll$ pénalité d'échec
- $AMAT = HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1}$
- $MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}$
- $AMAT = HitTime_{L1} + MissRate_{L1} \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2})$



Cohérence entre les niveaux de cache

- Propriété d'inclusion entre les niveaux de cache :
- **Inclusif** : les lignes contenues dans le cache L1 sont aussi contenues dans le cache L2
  - Exemple : ARM Cortex A53 pour les instructions
- **Exclusif** : les lignes sont contenues soit dans le cache L1, soit dans le cache L2
  - Exemple : AMD Athlon XP, ARM Cortex A53 pour les données
- Avantages/inconvénients :
  - Taille de stockage effective = taille du cache L2 (inclusif, -), ou taille du cache L1 + taille du cache L2 (exclusif, +)
  - En cas de miss L1 hit L2, copie de la ligne du L2 vers le L1 (inclusif, +), ou échange de ligne entre le L1 et le L2 (exclusif, -)
  - Une éviction du L2 peut provoquer des évictions dans le L1 (inclusif, -)
  - Une éviction de toute la hiérarchie de cache doit seulement vérifier le cache L2 (inclusif, +, puis mécanisme géré par le L2) ou tous les caches (exclusif, -)
  - Les lignes de cache du L2 peuvent être plus grandes que celles du L1 (inclusif, +) ou doivent être de la même taille (exclusif, -)
- **Non inclusif** : les lignes contenues dans le cache L1 peuvent être contenues dans le cache L2, mais pas forcément
  - Taille de stockage effective (maximale) = taille du cache L1 + taille du cache L2
  - Exemple : Intel Core IvyBridge
- Même problématique d'inclusivité entre le L2 et le L3

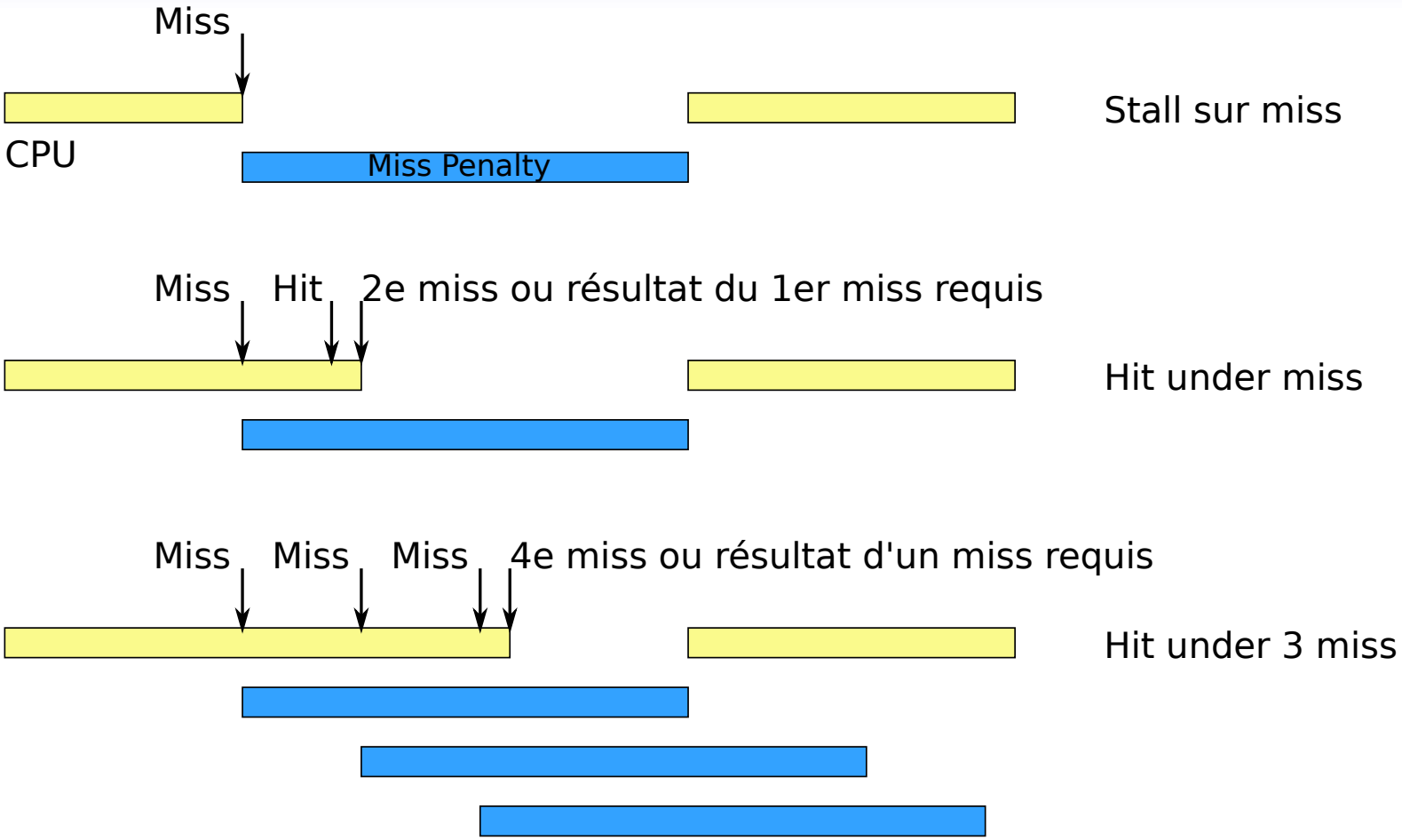
Redémarrage précoce et mot demandé en premier

- Idée : ne pas attendre que tout le bloc mémoire soit chargé pour utiliser les données (répondre au processeur)
- **Redémarrage précoce** : dès que la donnée nécessaire est chargée, l'utiliser
- **Mot demandé en premier** : charger en premier la donnée ayant produit l'échec
- Peut s'appliquer à l'intérieur du cache uniquement (lors de sa mise à jour), ou dans le protocole de communication entre le L1 et le L2
- En général, utile quand la taille des blocs est grande

Cache non-bloquant

- Permettre au processeur de continuer son exécution après un miss
  - $\Rightarrow$ Lors d'un accès à un registre, vérifier qu'il n'y a pas de miss en attente dessus
- Le cache peut continuer à être accédé même lorsqu'il est en train de traiter un miss
- *Hit under miss* : le cache peut traiter les requêtes qui font *hit* quand il y a un miss en cours, les requêtes qui font à nouveau miss sont mises en attente
- *Hit under multiple miss* : le cache peut traiter les requêtes qui font *hit* quand il y a plusieurs (exemple : 4) miss en cours
  - Le cache doit gérer l'envoi de plusieurs requêtes en parallèle et la réception des réponses possiblement dans le désordre
  - Complexe
- Permet de masquer la latence d'accès à la mémoire (ou au cache L2)
- Exemple : Pentium Pro

Cache non-bloquant : illustration

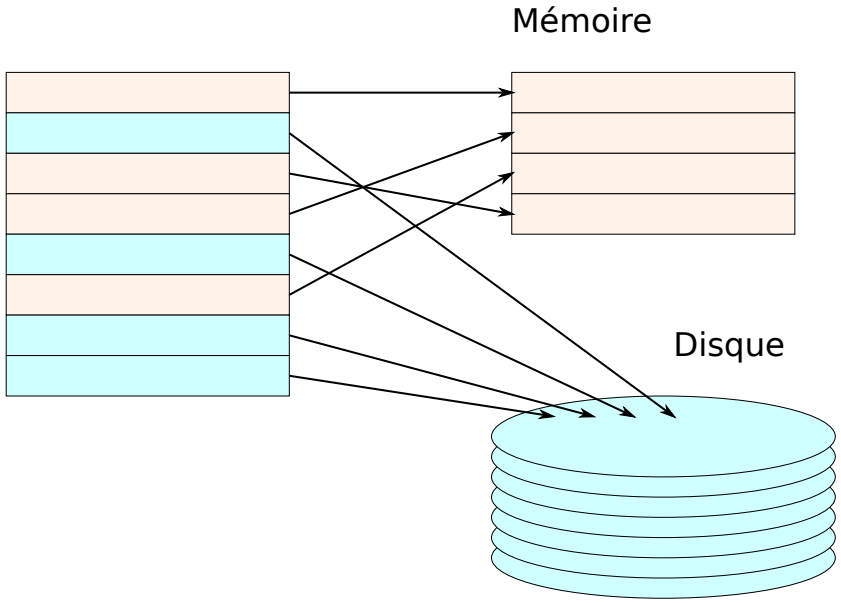


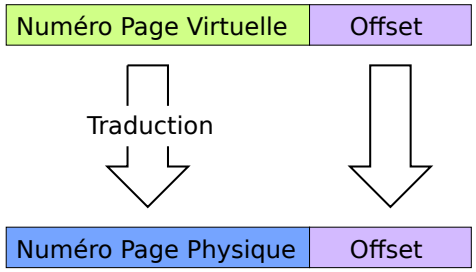
Performances des caches non-bloquant

- Permettre l'accès au cache quand il y a 1 ou 2 miss permet de gagner significativement sur le temps passé à attendre les réponses aux miss (temps de *stall*)
- Peu de gain au-delà

Accélerer la traduction d'adresse

- Avantages de la mémoire virtuelle
  - Compilation indépendante des programmes
  - Isolation des processus
  - Gestion des droits/Protection
  - Utiliser une partie du disque comme extension de la mémoire vive (swap)
- MMU (*Memory Management Unit*) : Matériel effectuant la traduction des adresses



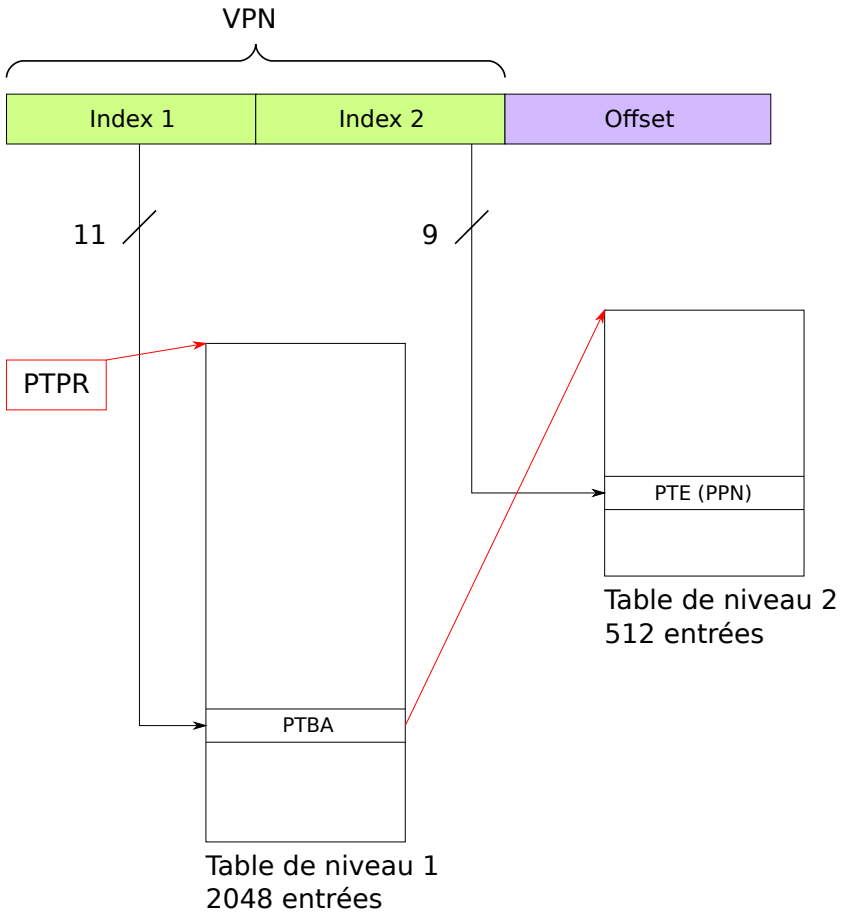


Accélerer la traduction d'adresse

- Analogie entre le cache L1 et le mécanisme de traduction d'adresse

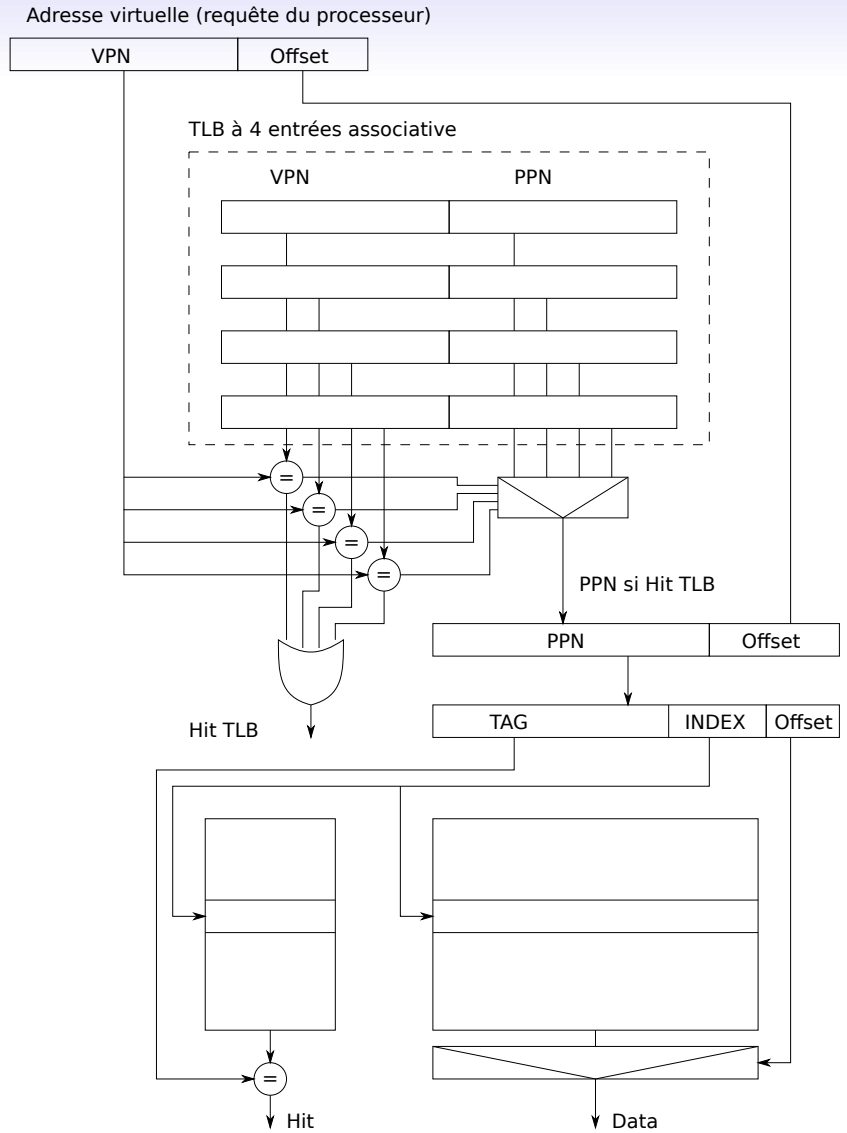
Paramètre	Cache L1	Mémoire Virtuelle
Taille du bloc	16B – 128B	4096B – 2MB
Temps d'accès (Hit)	1 – 2 cycles	**40 – 150 cycles**
Pénalité d'échec	10 – 100 cycles	1M – 10M cycles
Taux d'échec	10%	0.001%

- Temps d'accès de la mémoire virtuelle prohibitif : de l'ordre de grandeur de la pénalité d'échec du cache L1
- $\Rightarrow$ Utilisation d'un cache de traduction : TLB



Accélerer la traduction d'adresse : accès au cache avec TLB

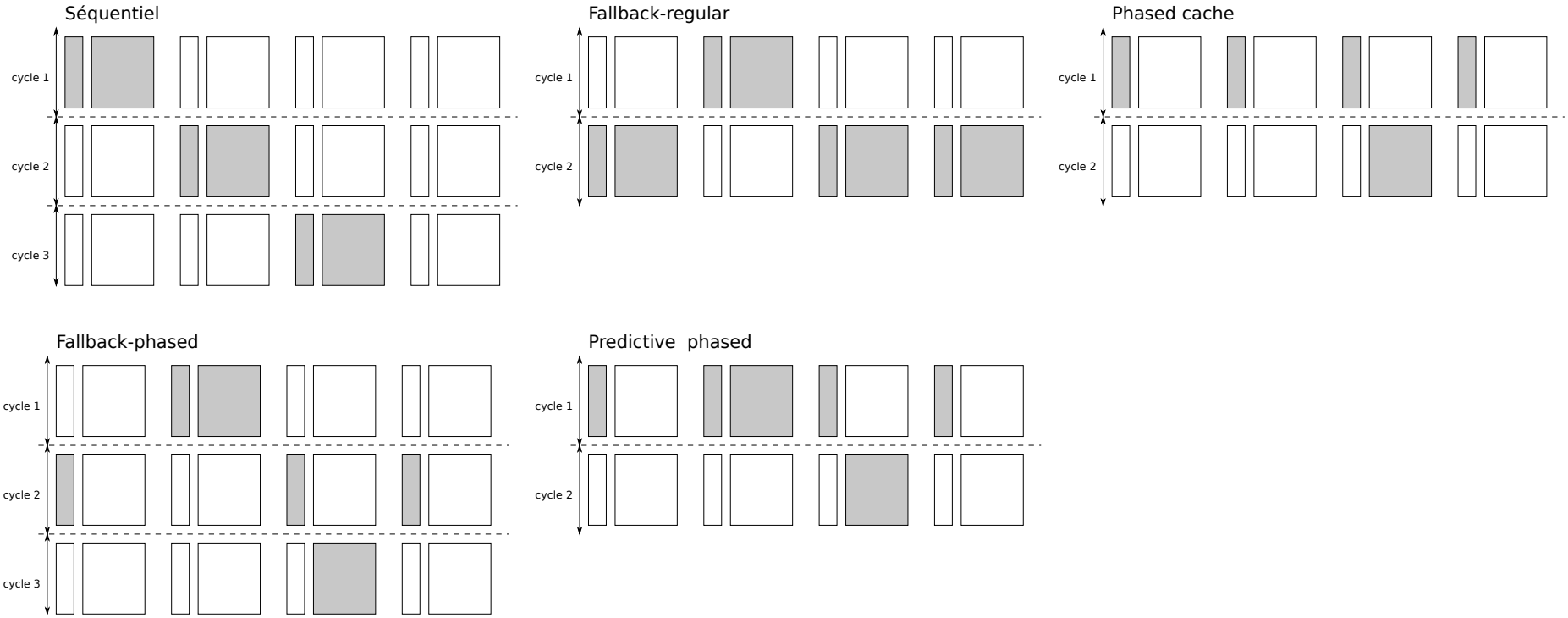
- TLB (*Translation Lookaside Buffer*) : cache qui contient les traductions des dernières pages accédées (1 traduction = 1 entrée de la table des pages)
- Cache classique avec un haut degré d'associativité (peu de pages différentes sont accédées par un programme, mais on veut à tout prix éviter les conflits)
- TAG TLB + Index TLB = Numéro de page virtuelle (différents des TAG et index du cache)
- Donnée = Numéro de page physique (+ droits)
- Souvent, présence de deux TLB : une pour les instructions et une pour les données
- Peut rallonger le chemin critique



Réduire le temps d'accès au cache

- Temps d'accès : temps d'accès aux données + temps pour déterminer si c'est un succès ou un échec (accès aux meta-données)
  - Directement lié à la taille du cache
  - Augmente avec la fréquence
  - Augmente avec l'associativité
- Caches petits et simples
- Pipeliner l'accès au cache
- $\Rightarrow$ Différentes politiques d'accès plus ou moins séquentielles (même idée que pour les caches pseudo-associatifs)

Réduire le temps d'accès au cache : différentes politiques



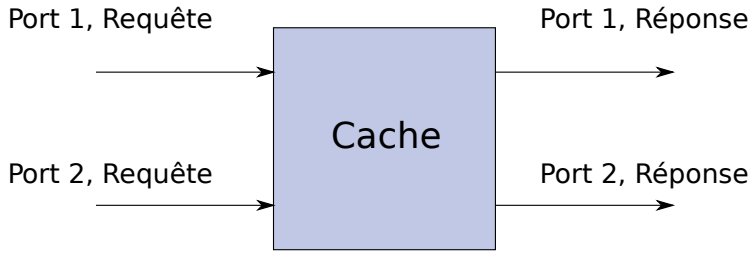
- Intel Pentium 4 : Fallback-Regular (consommation)

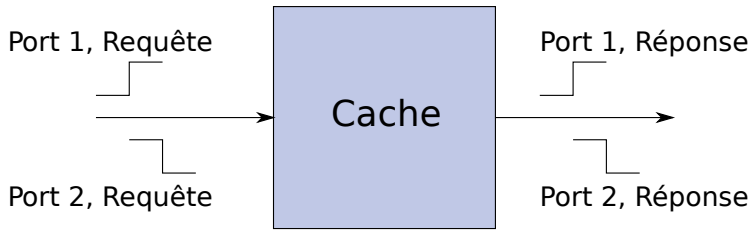
Augmenter le débit d'instructions chargées

- L'unité de chargement doit assurer un débit suffisant pour l'ILP
  - Exemples : superscalaire, chargement de données en parallèle
  - Couplé à la prédiction de branchement
- Chargement de plusieurs instructions/données : caches multi-ports
  - Multi-ports
  - *Overclocking*
  - Cache multiple
  - Partitions en bancs
- Une combinaison entre cache et predicteur de branchement : le trace cache

Caches multi-ports

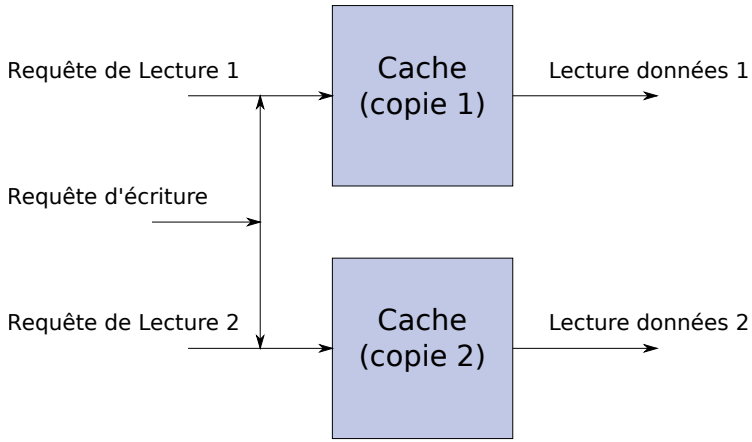
- Multi-port : 2 ports accessibles en parallèle
  - Augmentation importante de la taille du cache et du temps d'accès en cas de hit
  - Doit contenir une unité de résolution des conflits (deux requêtes ne peuvent pas accéder simultanément aux mêmes entrées du répertoire et du cache)
- *Overclocking*
  - Horloge du cache 2 fois plus rapide que celle du processeur
  - Un accès lors du front montant et un lors du front descendant de l'horloge



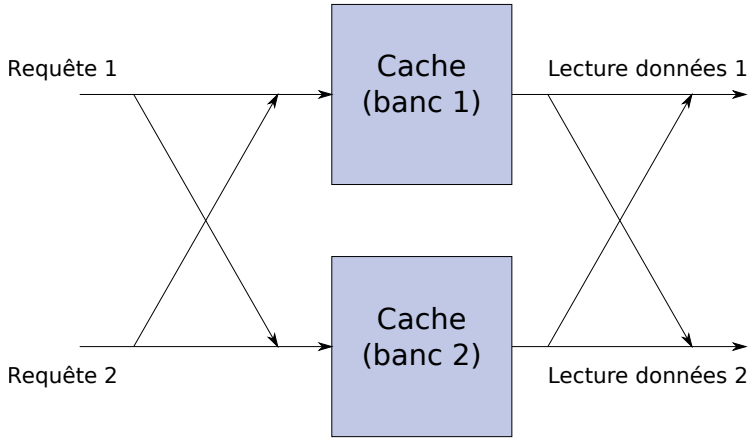


Caches multi-ports

- Cache multiple : 2 load / 1 store
  - Taille du cache multipliée par 2



- Partition des adresses en plusieurs bancs
  - Conflits potentiels entre les bancs
  - Pas d'accès multiples à un banc



Le Trace Cache

- Cache qui capture les séquences dynamiques d'instructions
- Contient des suites d'instructions contenant des branchements, potentiellement pris (instructions B, D et G)
- Tester si l'accès fait hit nécessite de connaitre le résultat du potentiel branchement et de le comparer à la valeur stockée pour la ligne