Architecture des Ordinateurs Avancée (L3)

Cours 3 - Le cache

Carine Pivoteau [1]

1. En collaboration avec X. Goaoc

Retour sur le TP 2 : mesurer le temps d'exécution

Le temps CPU consommé par le programme correspond au temps d'activité consacré par la machine à l'exécution du programme. Sur une machine multi-cœurs, ce temps peut être plus grand que le temps écoulé si plusieurs cœurs sont dédiés à la tâche.

- La commande `time` : affiche le temps écoulé et le temps CPU. Simple mais ne permet de mesurer qu'un programme *complet*.
- Sous linux, la commande `perf` fournit un diagnostic assez complet, avec notamment la lecture de plusieurs compteurs du CPU (dont le compteur de cycles lu par `rdtsc`).

▷ Mais ces commandes ne permettent pas de mesurer juste une portion du programme...

Retour sur le TP 2 : mesurer le temps d'exécution

Plusieurs bibliothèques fournissent des fonctions qui relèvent le temps depuis le début du programme, sous différentes formes. Ainsi, on peut **chronométrer** une partie d'un programme C en modifiant son code.

- `time()` : résolution d'une seconde :(
- `clock()` : relève le temps CPU dédié au programme courant (système UNIX). La résolution de cette horloge dépends de la machine et peut être déterminée via la constante `CLOCKS_PER_SEC` :
  https://en.cppreference.com/w/c/chrono/clock.
  Attention, sous Windows, elle mesure le temps réel :
  https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/clock.
- Le CPU incrémente un compteur à intervalles réguliers. On peut lire sa valeur via l'instruction assembleur `rdtscp` ou, en C, via la commande `__rdtscp()` de la bibliothèque `<x86intrin.h>`. Elle mesure les cycles d'horloge de façon plus fine que `clock()` et permet d'éviter certains problèmes liés à l'execution *out-of-oder* (voir suir du cours), mais n'est pas tès portable.

Retour sur le TP 2 : mesurer “correctement”

Chronométrer plusieurs fois le même code peut produire des résultats assez variables, en temps écoulé comme en temps CPU.

- Les conditions d'utilisation du CPU peuvent varier d'une exécution à l'autre, avec par exemple le réveil de certaines tâches de fond.
- La fréquence du processeur s'adapte à la charge de travail de manière à réduire la consommation au repos (cf commande linux `cpupower`). Le même code peut donc être plus rapide une fois le processeur “réveillé”.
- Le préchargement du programme ou des données en mémoire cache peut grandement influer la rapidité d'exécution d'un code.
- Le temps d'exécution peut être trop court pour être significatif.
- ...

Pour nos expériences, on prend deux précautions simples :

1. Appeler la fonction plusieurs fois avant de commencer à mesurer.
2. Ensuite, chronométrer le temps moyen d'exécution pour de nombreuses exécutions.

Retour sur le TP 2

```
void print_time(int n, int (*f_to_time)(int), int count){
  int sum = 0; // Penser à utiliser le résultat du calcul...
  unsigned long tic, toc;

  for (int i = 0; i < 3; ++i){ sum += f_to_time(n); } // préchauffage

  ////////////////////////////
  tic = clock();
  ////////////////////////////
  for (int i = 0; i < count; ++i){
    sum += f_to_time(n);
  }
  ////////////////////////////
  toc = clock();
  ////////////////////////////

  float time = (toc-tic)/(float)CLOCKS_PER_SEC/count;
  printf("%10d -- %.10f sec\n", sum, time);
}
```

Retour sur le TP 2

```
void print_cycles(int n, int (*f_to_time)(int), int count){
  int sum = 0; // Penser à utiliser le résultat du calcul...
  unsigned long tic, toc;
  unsigned int ui;

  for (int i = 0; i < 3; ++i){ sum += f_to_time(n); } // préchauffage

  ////////////////////////////
  tic = __rdtscp(&ui);
  ////////////////////////////
  for (int i = 0; i < count; ++i){
    sum += f_to_time(n);
  }
  ////////////////////////////
  toc = __rdtscp(&ui);
  ////////////////////////////

  printf("%10d -- %lu cycles\n", sum, (toc-tic)/count);
}
```

Fin du TP 2 : accès séquentiels VS accès aléatoires

Faire un grand nombre n d'accès dans un très grand tableau ($N = 10^9$) et comparer le temps pris quand ces accès sont consécutifs au temps pris quand ils sont aléatoires.

1. Comparer les temps d'exécution de `acces_seq` et `acces_alea` pour différentes valeurs de `n`.
2. Donner au moins deux raisons qui peuvent expliquer que ces temps sont différents.
3. Refaire vos mesures en commençant par préparer des tableaux auxiliaires contenant la liste des accès à effectuer.
4. Mesurez les temps d'accès séquentiels et aléatoires pour diverses valeurs de n.
5. Utilisez les tests que vous venez de faire pour estimer la taille du cache de dernier niveau.

L'organisation du cache

On reprend : exemple de cache à adressage direct

Prenons l'exemple d'un système avec les caractéristiques suivantes :

- On choisit une **mémoire principale (RAM) de 4 Go**. Quelle est la taille d'adresse minimale nécessaire pour pouvoir accéder à la totalité de cette mémoire ?
- On utilise un **cache de 32 Ko**, avec des **lignes de 64 octets**. Combien de lignes y a-t-il dans le cache ?
- Comment faire le mapping des adresses de la RAM vers le cache simplement/efficacement ?

On reprend : exemple de cache à adressage direct

Prenons l'exemple d'un système avec les caractéristiques suivantes :

- On choisit une **mémoire principale (RAM) de 4 Go**. Quelle est la taille d'adresse minimale nécessaire pour pouvoir accéder à la totalité de cette mémoire ?
  $\triangleright$ 4 Go $= 2^{32}$ octets, donc des adresses de 32 bits.
- On utilise un **cache de 32 Ko**, avec des **lignes de 64 octets**. Combien de lignes y a-t-il dans le cache ?

- Comment faire le mapping des adresses de la RAM vers le cache simplement/efficacement ?

On reprend : exemple de cache à adressage direct

Prenons l'exemple d'un système avec les caractéristiques suivantes :

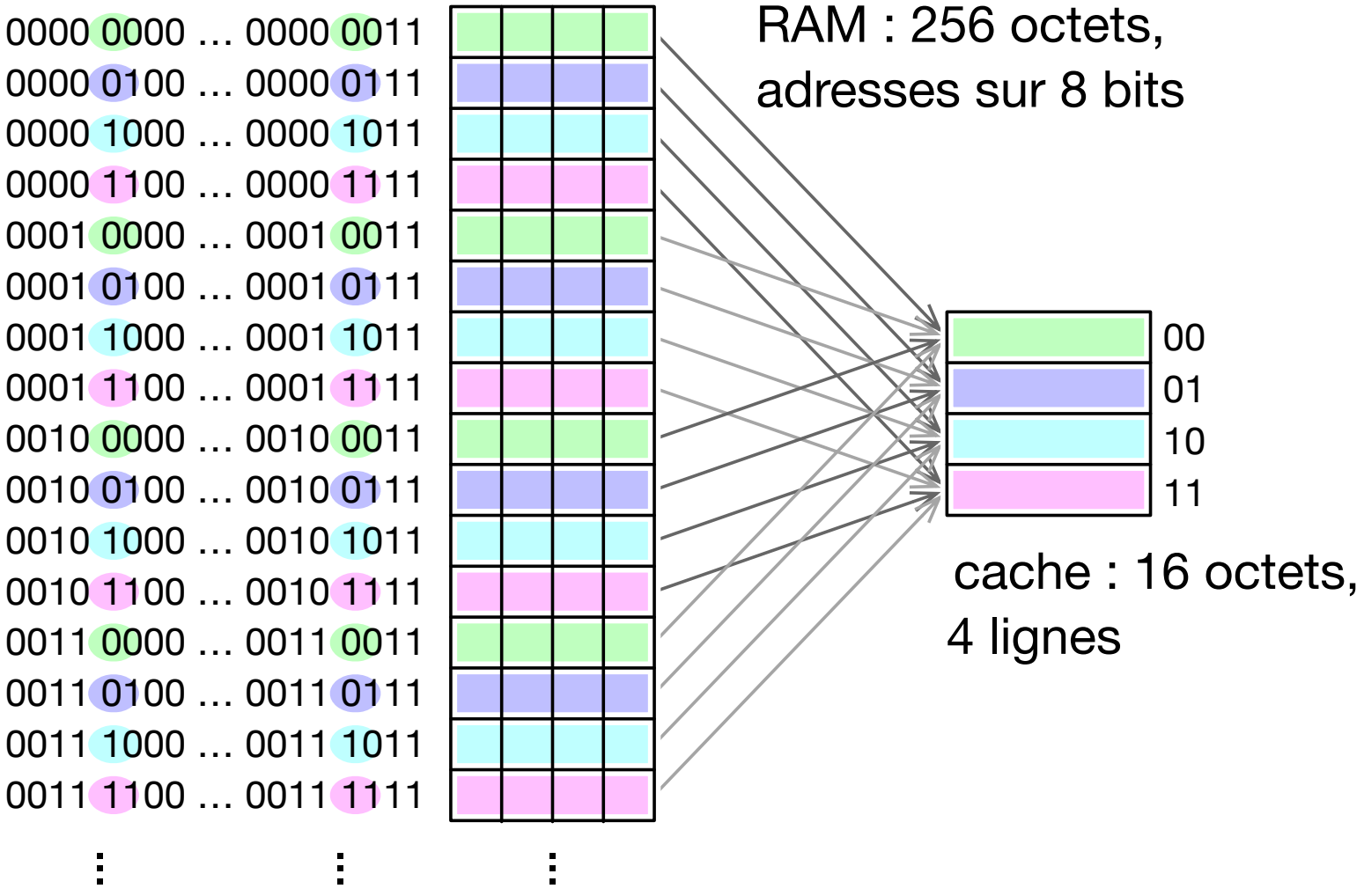
- On choisit une **mémoire principale (RAM) de 4 Go**. Quelle est la taille d'adresse minimale nécessaire pour pouvoir accéder à la totalité de cette mémoire ?
  $\triangleright$ 4 Go $= 2^{32}$ octets, donc des adresses de 32 bits.
- On utilise un **cache de 32 Ko**, avec des **lignes de 64 octets**. Combien de lignes y a-t-il dans le cache ?
  $\triangleright$ 32 Ko $= 2^{15}$ octets, $64 = 2^6$. nb lignes $= 2^{15}/2^6 = 2^9 = 512$.
- Comment faire le mapping des adresses de la RAM vers le cache simplement/efficacement ?

On reprend : exemple de cache à adressage direct

Prenons l'exemple d'un système avec les caractéristiques suivantes :

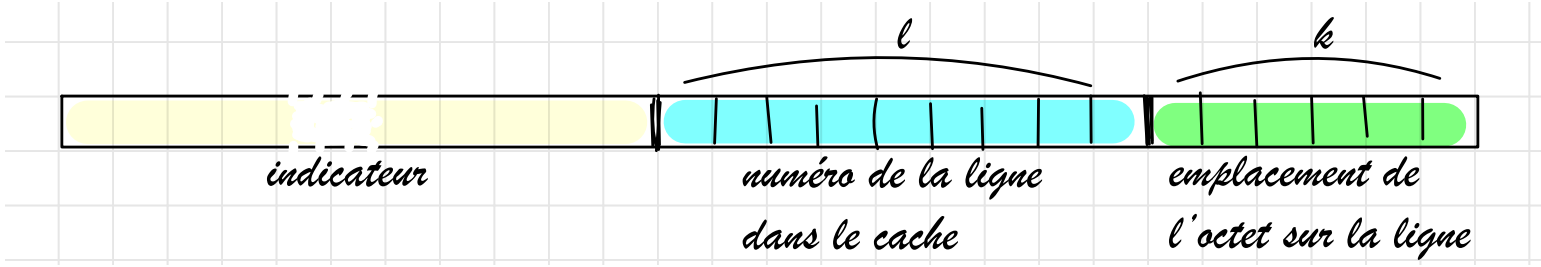
- On choisit une **mémoire principale (RAM) de 4 Go**. Quelle est la taille d'adresse minimale nécessaire pour pouvoir accéder à la totalité de cette mémoire ?
  $\triangleright$ 4 Go $= 2^{32}$ octets, donc des adresses de 32 bits.
- On utilise un **cache de 32 Ko**, avec des **lignes de 64 octets**. Combien de lignes y a-t-il dans le cache ?
  $\triangleright$ 32 Ko $= 2^{15}$ octets, $64 = 2^6$. nb lignes $= 2^{15}/2^6 = 2^9 = 512$.
- Comment faire le mapping des adresses de la RAM vers le cache simplement/efficacement ?
  Autrement dit, comment calculer le numéro (codé sur 9 bits) de la ligne de cache correspondant à une adresse 32 bits ?

Mapping des adresses de la RAM vers le cache



Calcul de l'emplacement en cache.

- Avec un cache de 2^ℓ **lignes**, chacune contenant 2^k **octets**.
- Découpage de l'adresse mémoire pour obtenir le numéro de ligne :



- 1 seule ligne possible dans le cache pour une adresse donnée.
- plusieurs blocs de la RAM correspondent à la même ligne de cache.
- Algo :
  1. trouver la ligne de cache,
  2. vérifier si elle est valide,
  3. vérifier l'indicateur ; en cas d'échec, remplacement de la ligne.

Exo : mini-cache

- Supposons que nous avons un cache de 32 octets dont les lignes font 4 octets. Et supposons que les adresses mémoires sont codées sur 8 bits.
- Quelle taille peut atteindre la mémoire adressable ?
- Combien de lignes y a-t-il dans le cache ?
- Quel est la taille de l'indicateur ?
- Où sera stockée la donné d'adresse 0011 0110 ?
- Quelles autres données seront stockées en cache en même temps ?
- Combien de cases de la mémoire principale sont susceptibles d'être stockées à la même adresse ?

Exo : collisions

- Proposer une expérience permettant de **mettre en évidence l'adressage direct dans le cache**.
- Il suffit de comparer les temps d'exécution des 2 boucles suivantes sur un très grand tableau :
  - une boucle qui accède un très grand nombre de fois à des cases espacées d'une grande puissance de 2 (par exemple la taille du dernier cache multiplié par `sizeof(int)`),
  - une boucle qui accède un très grand nombre de fois à des cases espacées d'une valeur proche de cette grande puissance de 2, mais qui n'est pas une puissance de 2.
  - Comparer avec une boucle qui fait des accès séquentiels.

(fichier **demo-C3-1.c**)

Exo : collisions

- Proposer une expérience permettant de **mettre en évidence l'adressage direct dans le cache**.
- Il suffit de comparer les temps d'exécution des 2 boucles suivantes sur un très grand tableau :
  - une boucle qui accède un très grand nombre de fois à des cases espacées d'une grande puissance de 2 (par exemple la taille du dernier cache multiplié par `sizeof(int)`),
  - une boucle qui accède un très grand nombre de fois à des cases espacées d'une valeur proche de cette grande puissance de 2, mais qui n'est pas une puissance de 2.
  - Comparer avec une boucle qui fait des accès séquentiels.

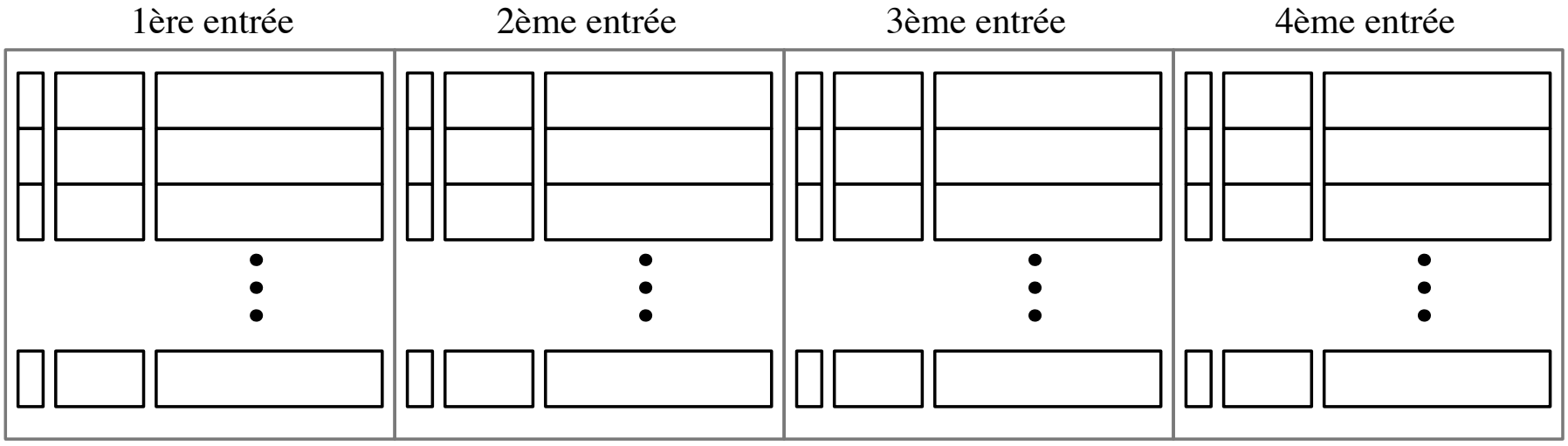
(fichier **demo-C3-1.c**)

Exo : collisions

- Proposer une expérience permettant de **mettre en évidence l'adressage direct dans le cache**.
- Il suffit de comparer les temps d'exécution des 2 boucles suivantes sur un très grand tableau :
  - une boucle qui accède un très grand nombre de fois à des cases espacées d'une grande puissance de 2 (par exemple la taille du dernier cache multiplié par `sizeof(int)`),
  - une boucle qui accède un très grand nombre de fois à des cases espacées d'une valeur proche de cette grande puissance de 2, mais qui n'est pas une puissance de 2.
  - Comparer avec une boucle qui fait des accès séquentiels.

(fichier **demo-C3-1.c**)

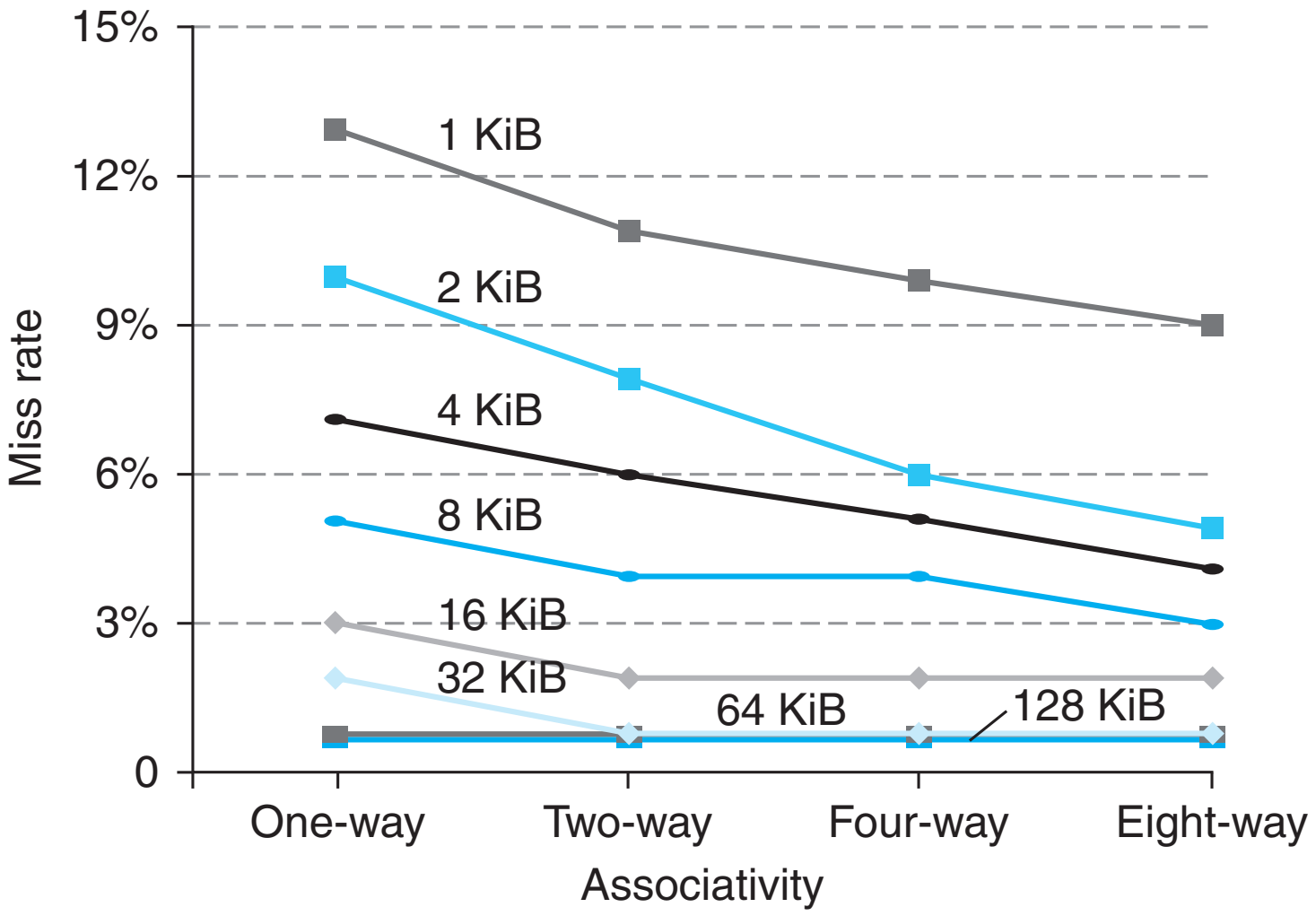
Cache associatif



- Chaque entrée du cache contient *plusieurs blocs mémoire* (ici 4) en même temps.
- Besoin d'un algorithmes de remplacement des lignes de cache pour **choisir la ligne à effacer** si une ligne est manquante. Dans l'idéal, on voudrait celle qui risque d'être réutilisée le plus tard.
- Autre question : à quel moment faire la mise à jour en mémoire ?

L'effet de l'associativité sur le nombre de cache miss

source : Patterson et Hennessy, Computer Organization and Design



Récapitulatif

- Adressage direct non associatif :
  - Chaque adresse de la mémoire correspond à un seul emplacement dans le cache. Si deux données différentes doivent être stockée au même endroit, l'accès à l'une éjectera l'autre : on a une collision.
  - Pas de politique de remplacement "intelligente".
- k-associatif :
  - On peut stocker k données à la même adresse dans le cache.
  - On peut "choisir" quelle donnée va sortir du cache.
- Complètement associatif (*fully associative*)
  - Toutes les données sont stockées à la même adresse dans le cache, c'est à dire n'importe où.
  - Le cache est entièrement contrôlé par la politique de remplacement.

Compromis : Une collision (*cache miss*) peut coûter très cher mais l'algorithme de remplacement prend du temps aussi.

Exo : mini-cache (suite)

- Reprenons le cache de 32 octets dont les lignes font 4 octets.
- Dessiner un tel cache 2-associatif.
- Où sera stockée la donné d'adresse 0011 0110 ?
- Quelles autres données peuvent être stockées au même endroit ?
- Mêmes questions avec un cache *fully-associative*.

Un peu d'algorithmique...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)

- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
  - compteurs (hardware)
  - move-to-front (algo) : table de hachage + liste doublement chaînée
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
  - compteurs (hardware)
  - move-to-front (algo) : table de hachage + liste doublement chaînée
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
  - compteurs (hardware)
  - move-to-front (algo) : table de hachage + liste doublement chaînée
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
  - compteurs (hardware)
  - move-to-front (algo) : table de hachage + liste doublement chaînée
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
  - compteurs (hardware)
  - move-to-front (algo) : table de hachage + liste doublement chaînée
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique de remplacement d'une ligne de cache

- Optimal
- First in first out (FIFO) : la file d'attente
- Least Recently Used (LRU)
  - compteurs (hardware)
  - move-to-front (algo) : table de hachage + liste doublement chaînée
- Most Recently Used (MRU)
- Pseudo LRU (PLRU) : 1 seul bit d'information
- Random
- Least Frequently Used (LFU)
- ...

Politique d'écriture dans la mémoire de niveau supérieur

Le cache sert de *buffer* intermédiaire entre le processeur et la mémoire. Pour que cela ait un intérêt, il faut que le cache reste "à jour".

Autrement dit, quand on change la valeur d'une donnée qui est en cache, on va évidemment changer cette valeur en cache...

... mais puisqu'elle pourra directement être lue en cache, pourquoi l'écrire en mémoire principale ?

Politique d'écriture dans la mémoire de niveau supérieur

Le cache sert de *buffer* intermédiaire entre le processeur et la mémoire. Pour que cela ait un intérêt, il faut que le cache reste "à jour".

Autrement dit, quand on change la valeur d'une donnée qui est en cache, on va évidemment changer cette valeur en cache...

... mais puisqu'elle pourra directement être lue en cache, pourquoi l'écrire en mémoire principale ?

- Elle pourrait ne plus être en cache (remplacée par une autre),

Politique d'écriture dans la mémoire de niveau supérieur

Le cache sert de *buffer* intermédiaire entre le processeur et la mémoire. Pour que cela ait un intérêt, il faut que le cache reste "à jour".

Autrement dit, quand on change la valeur d'une donnée qui est en cache, on va évidemment changer cette valeur en cache...

... mais puisqu'elle pourra directement être lue en cache, pourquoi l'écrire en mémoire principale ?

- Elle pourrait ne plus être en cache (remplacée par une autre),
- Un autre thread pourrait en avoir besoin (cf cours de concurrence, l'année prochaine).

Politique d'écriture dans la mémoire de niveau supérieur

Le cache sert de *buffer* intermédiaire entre le processeur et la mémoire. Pour que cela ait un intérêt, il faut que le cache reste "à jour".

Autrement dit, quand on change la valeur d'une donnée qui est en cache, on va évidemment changer cette valeur en cache...

... mais puisqu'elle pourra directement être lue en cache, pourquoi l'écrire en mémoire principale ?

- Elle pourrait ne plus être en cache (remplacée par une autre),
- Un autre thread pourrait en avoir besoin (cf cours de concurrence, l'année prochaine).

Donc la questions, c'est plutôt : **à quel moment doit-on l'écrire en mémoire principale ?** C'est la question de la *Write policy*.

Write policy - compromis - performances

Quand on écrit une donnée dans le cache, il y a deux options pour la mise à jour de la mémoire principale :

- on écrit **immédiatement** *(write-through)* :
  - cohérence facile à maintenir, pas besoin d'un bit d'information
  - écriture lente
  - beaucoup d'écritures inutiles
- on écrit **plus tard** *(write-back)*, quand elle sera évincée du cache :
  - utilisation d'un *dirty* bit
  - moins d'écritures en mémoire principale
  - *read-miss* plus coûteux (il faut d'abord écrire en RAM).

Ce choix impacte les **performances du cache** !

Write policy - compromis - performances

Quand on écrit une donnée dans le cache, il y a deux options pour la mise à jour de la mémoire principale :

- on écrit **immédiatement** *(write-through)* :
  - cohérence facile à maintenir, pas besoin d'un bit d'information
  - écriture lente
  - beaucoup d'écritures inutiles
- on écrit **plus tard** *(write-back)*, quand elle sera évincée du cache :
  - utilisation d'un *dirty* bit
  - moins d'écritures en mémoire principale
  - *read-miss* plus coûteux (il faut d'abord écrire en RAM).

Ce choix impacte les **performances du cache** !

... et si la donnée à écrire n'est pas déjà dans le cache (*write-miss*) ?

Write policy

Quand la donnée à écrire n'est pas déjà dans le cache (write-miss), ...

- est-ce qu'**on l'écrit en cache** (*write allocate*) ?
  ▷ Dans ce cas, la donnée est chargée en cache (on a un *cache-read miss*) et puis écrite (*cache-write hit*).
- ou **seulement en mémoire principale** (*no-write allocate*) ?

Write policy

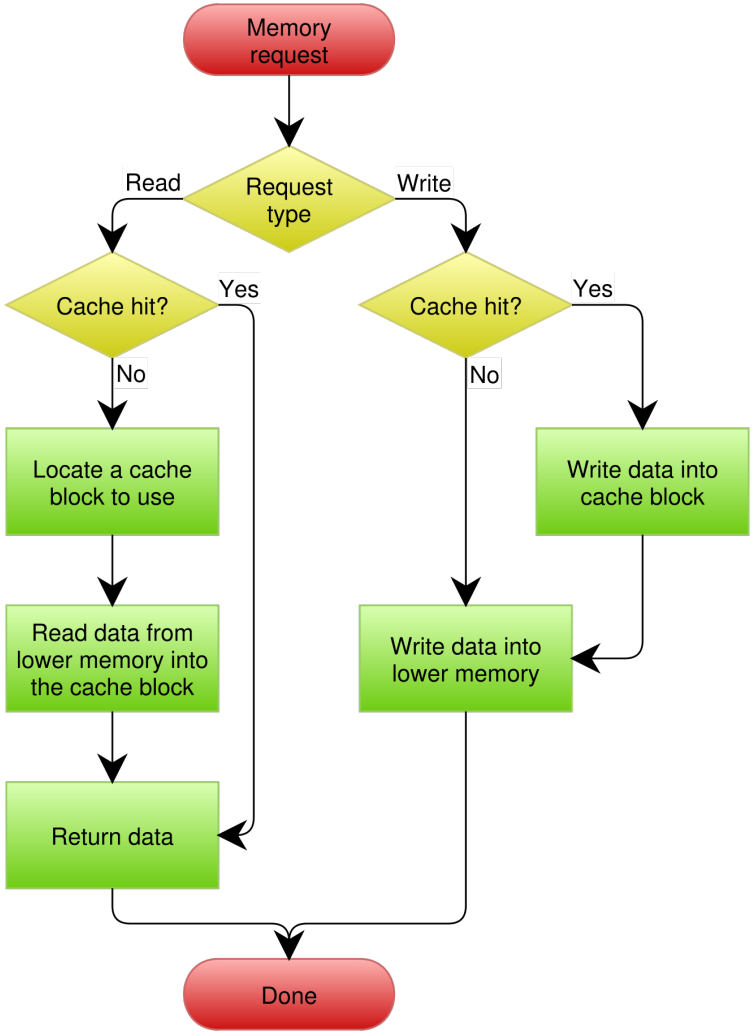
Quand la donnée à écrire n'est pas déjà dans le cache (write-miss), ...

- est-ce qu'**on l'écrit en cache** (*write allocate*) ?
  ▷ Dans ce cas, la donnée est chargée en cache (on a un *cache-read miss*) et puis écrite (*cache-write hit*).
- ou **seulement en mémoire principale** (*no-write allocate*) ?

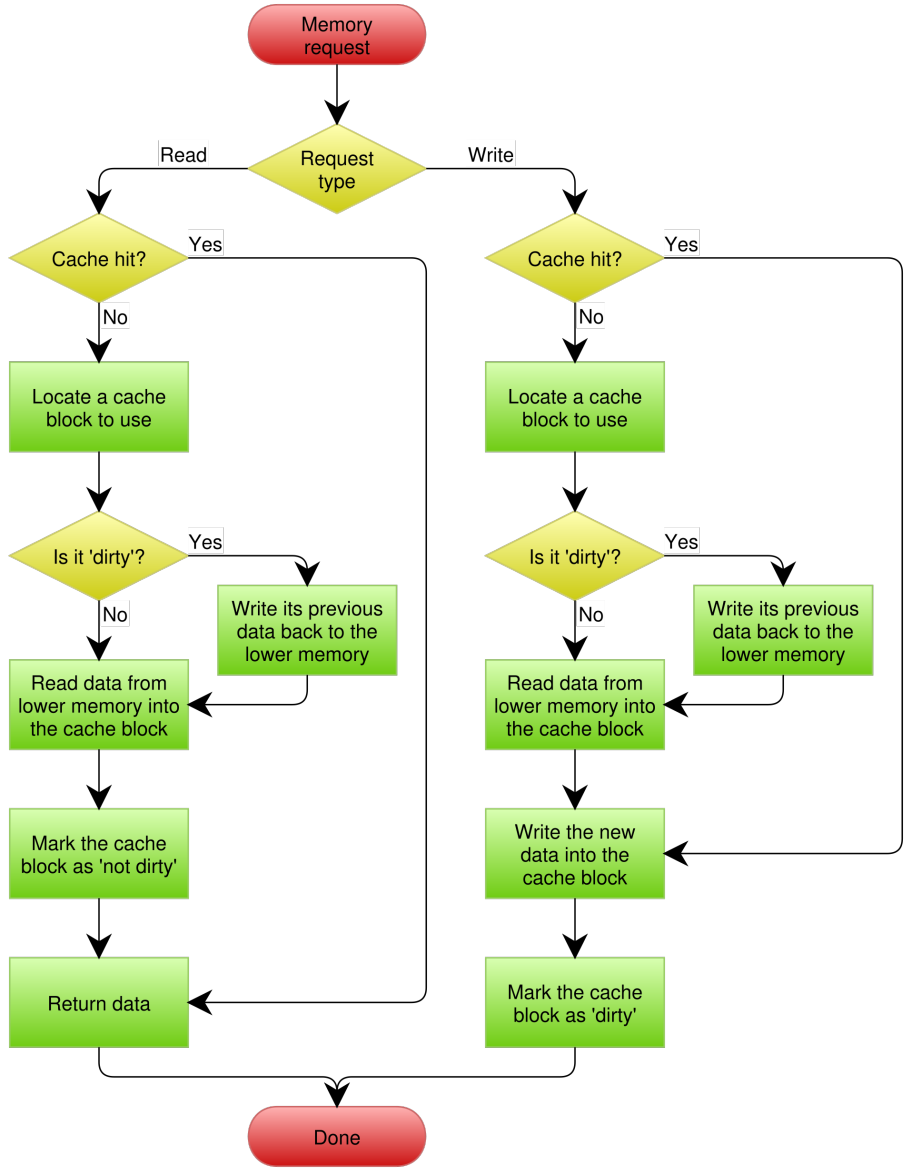
En général, on combine :

- **Write-through** avec **no-write allocate**
  ▷ Politique simple mais qui fait beaucoup d'allers-retours avec la mémoire principale.
- **Write back** avec **write allocate**
  ▷ Politique plus complexe, mais qui exploite mieux la localité temporelle, et donc plus efficace.

Write-through + no-write allocation



Write-back + write allocation



source : Wikipédia

Bonus

Et dans l'autre sens ?

Jusque là, on se pose la question d'améliorer le temps d'accès à mémoire. Et pour cela, on est prêt à faire des sacrifices sur sa taille.

Mais si le problème que l'on rencontre, c'est l'inverse : on voudrait plus de mémoire, quitte à ce qu'elle soit plus lente ?

Et dans l'autre sens ?

Jusque là, on se pose la question d'améliorer le temps d'accès à mémoire. Et pour cela, on est prêt à faire des sacrifices sur sa taille.

Mais si le problème que l'on rencontre, c'est l'inverse : on voudrait plus de mémoire, quitte à ce qu'elle soit plus lente ?

- Solution = mémoire virtuelle.
- Utiliser de l'espace de stockage de masse pour étendre la mémoire vive : on peut sauvegarder sur un disque une portion de mémoire non utilisée "pour l'instant".
- On se pose alors des questions similaires à celles rencontrées pour la mise en place des caches : adressage, chargement des pages, politique de remplacement, compromis, ...

Tenir compte des effets de cache dans son code

On peut concevoir des algorithmes de façon à exploiter les caractéristiques du cache (taille, nombre/taille de ligne, ...)

- Algorithmes en mémoire externe (ou *cache-aware*) : besoin de connaître les paramètres du cache pour être efficaces.
  ▷ Exemple : *loop tiling* en algèbre
- Algorithmes (ou structures de données) *cache-oblivious* : efficaces quels que soient les paramètres du cache et le nombre de niveaux.
  - multiplication/transposition de matrices
  - static search trees
  - funnel sort
  - ...

`https://www.youtube.com/watch?v=50Ihl3hLflk`
`https://www.youtube.com/watch?v=Aa-BfOy2O3I`

`https://catonmat.net/mit-introduction-to-algorithms-part-fourteen`