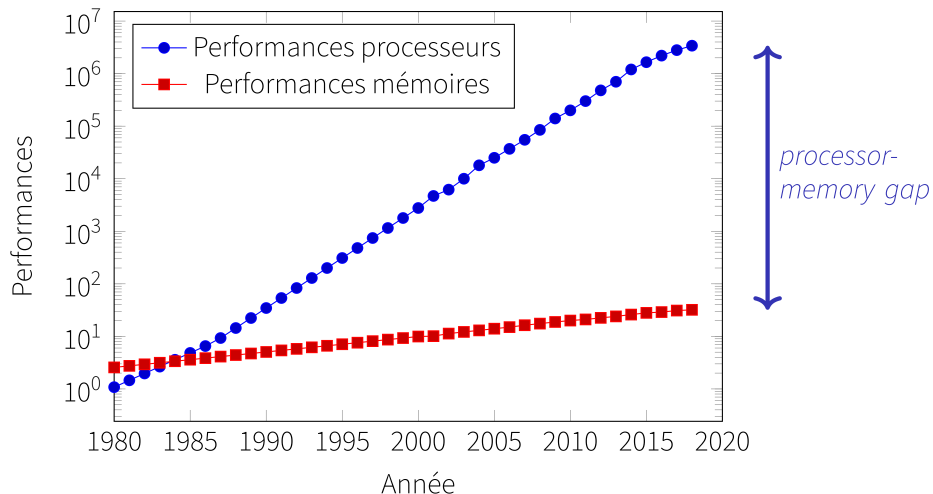
T1 Architecture des processeurs
Hiérarchie mémoire

A. Mérigot

M2 SETI 2022-23

Position du problème

Evolution technologique comparée des processeurs et des mémoires



Position du problème (cont.)

Les accès à la mémoire sont de plus très coûteux en énergie[1]

Opération	énergie (pJ)
add 8bits	0.03
add 32bits	0.1
mul 8bits	0.2
mul 32bits	3
add FP 16bits	0.4
add FP 32bits	0.9
mul FP 16bits	1
mul FP 32bits	4

Lecture 32 bits	énergie (pJ)
cache SRAM 8kB	10
cache SRAM 32kB	20
cache SRAM 1MB	100
DRAM	2000

[1] M. Horowitz ISSCC 2014

Principes de localité

Les accès à la mémoire ne sont pas quelconques.
Existence de « principes de localité » potentiellement exploitables pour accélérer les accès.

Localité temporelle

Si une information (instruction ou donnée) est manipulée par le processeur, il est très probable qu'elle sera également utilisée peu de temps après.

instructions d'une boucle, fonctions, variables locales, etc

Localité spatiale

Si une information est manipulée par le processeur, il est très probable qu'une information située dans une emplacement mémoire proche sera également utilisée peu de temps après.

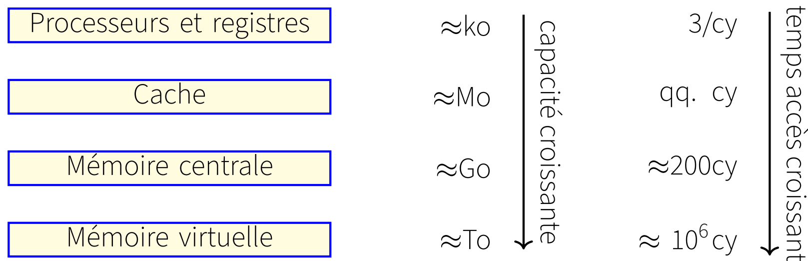
séquences d'instructions, élements d'un tableau, champs d'une struct, etc.

La hiérarchie mémoire

Les principes de localité permettent :

- d'identifier l'information utile (localité spatiale)
- de la transférer dans un dispositif rapide, mais de petite taille
- d'espérer que cette information sera réutilisée (localité temporelle)

Utilisation simultanée de différentes technologies de mémorisation organisées hiérarchiquement : registres, SRAM, DRAM, mémoire de masse.



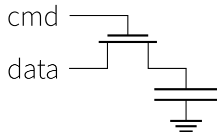
Les échanges entre les différents niveaux de la hiérarchie se font par *blocs*.

- cache ↔ mémoire centrale : *lignes de cache* (typ. 64o)
- mémoire centrale ↔ mémoire virtuelle : *pages* (typ. qq. ko)

Mémoires statiques et dynamiques

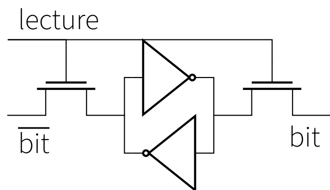
Deux grandes familles de mémoire : mémoires statiques et dynamiques.

mémoire dynamique (DRAM)



Information mémorisée dans une capacité (grille transistor).
Lecture destructrice nécessitant une réécriture.
La préservation de l'information nécessite un rafraichissement régulier (typ. 64ms)
utilisé comme mémoire principale

mémoire statique (SRAM)



Information mémorisée dans un *latch*
Temps de cycle fortement réduit
Surface + importante (6 trans./point)
utilisé dans les caches

Les mémoires dynamiques

Génération actuelle de DRAM :

DDR SDRAM

Double Data Rate

Ecrit les données sur les fronts
⌠ et ⌡ de l'horloge

Synchronous DRAM

Utilise une horloge H_m

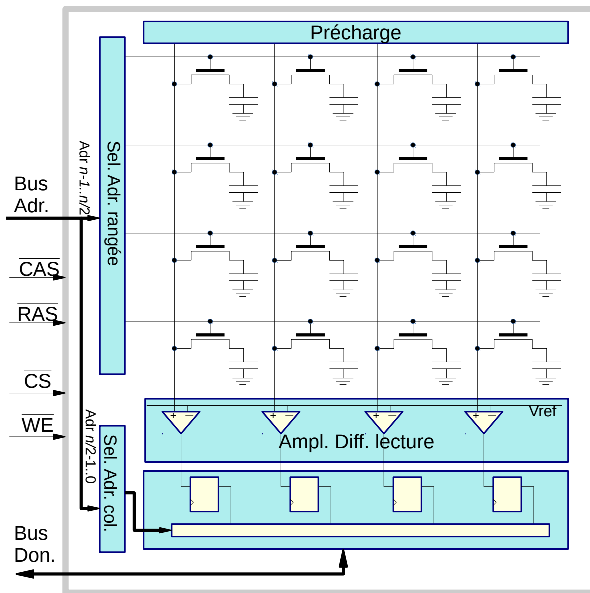
Capacité typique d'un circuit : 16-32 Gb

Horloge 800 Mhz – 1.6 GHz , typiquement 1GHz

E/S sur 64 bits (8o) par regroupement de plusieurs circuits dans un boîtier/module (DIMM) (*dual in line memory module*)

$\Rightarrow$ débit maximum 8o $\times$ 1G $\times$ 2 = 16 Go/s

Les mémoires dynamiques (cont.)



Mémoire dynamique
$4 \times 4 \times 1$ bit
(4bits d'adresse)

- les $n/2$ bits de poids forts de l'adresse sélectionnent la rangée,
- les $n/2$ bits de poids faible la colonne

Les mémoires dynamiques (cont.)

Les capacités de stockage des bits ont une valeur typique de 30 fF

Tension d'alimentation 1 V

Les capacités de bus par colonne sont environ 100× plus grandes.

⇒ très faible variation de tension (≈ 10 mV) à la lecture.

- avant la lecture, on précharge les bus à la moitié de la tension d'alimentation (≈ 0.5 V)
- ouverture des transistors de lecture
- lecture de la valeur du bit avec des amplificateurs différentiels (comparaison avec la tension de précharge)

La lecture est *destructive*. Nécessité de réécrire les valeurs lues.

Les mémoires dynamiques (cont.)

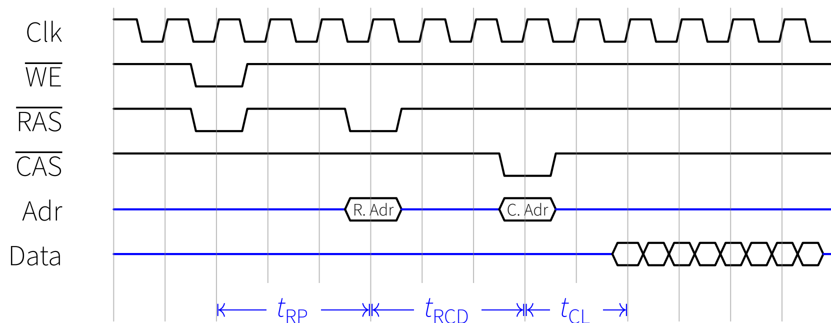
Processus de lecture :

1. envoi de la partie haute de l'adresse
   simultanément précharge des bus
   t_{RP} *RAS*[2] *Precharge Time*
2. arrêt de la précharge
   sélection de la ligne à lire
   lecture de la valeur dans les amplificateurs de lecture et mémorisation dans des latches de toute la rangée
   simultanément mémorisation de l'adresse de colonne
   t_{RCD} *RAS to CAS*[3] *Delay Time*
3. sélection de la colonne
   envoi de la donnée sur le bus de données
   simultanément, réécriture de la ligne dans les cases mémoire
   t_{CL} *CAS Latency*

[2] RAS : *Row Address Strobe*
[3] CAS : *Column Address Strobe*

Les mémoires dynamiques (cont.)

Chronogramme typique de lecture d'une mémoire dynamique DDR



Les mémoires dynamiques (cont.)

Valeurs typiques :

	temps(ns)	cycles mém. (1GHz)
t_{RP}	12–15 ns	12–15
t_{RCD}	12–15 ns	12-15
t_{CL}	10–12 ns	10–12

L'ensemble du cycle de lecture ($t_{RP}+t_{RCD}+t_{CL}$+envoi des données sur le bus) nécessite environ 40–50 cycles mémoire (à 1 GHz), soit 160–200 cycles processeur (à 4 GHz).

Si l'adresse de rangée (poids forts), ne change pas entre deux lectures on peut éviter de relire la rangée (*fast page mode*).

Le temps de lecture devient alors proche de t_{CL} ($\approx$ 50 cy processeur).

Les mémoires dynamiques (cont.)

Les écritures ont un mode de fonctionnement et des caractéristiques temporelles similaires.

L'écriture d'un mot impose

- la lecture de la rangée
- la modification du mot dans le registre de rangée
- la réécriture de la rangée

La combinaison des signaux $\overline{\text{CS}}$, $\overline{\text{RAS}}$, $\overline{\text{CAS}}$ et $\overline{\text{WE}}$ permet de définir des commandes de la mémoire (lecture, écriture, précharge, rafraichissement, etc) et de la configurer avec des paramètres.

Les mémoires dynamiques (cont.)

Les mémoires dynamique fonctionnent avec *accès en rafale* (*burst mode*).

Lors de la lecture d'une donnée, envoi des données environnantes dans la rangée, sans relecture de la ligne.

Actuellement (DDR4), *burst* de 8 données (en 4 cycles mémoire).

Si la largeur du bus est de 64 bits, chaque lecture envoie 64 octets en séquence.

Lors d'une demande de lecture à l'adresse binaire *yyyyy*...*yyyxxxxxx*, on envoie le contenu des cases *yyyy*...*yy*0000000 à *yyyy*...*yy*111111.

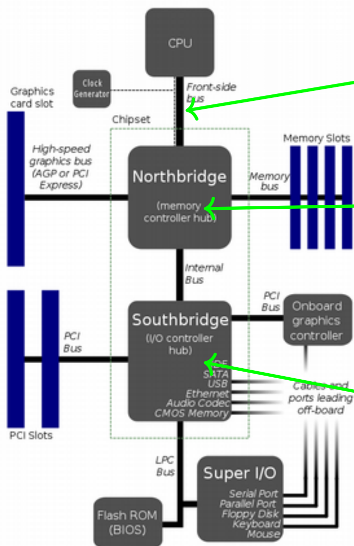
On peut (ou non) commencer par le mot à l'adresse donnée (*critical word first*).

Cette longueur de rafale doit être cohérente avec la taille d'une ligne de cache.

Les écritures peuvent être ou non en mode *burst*.

L'ensemble est géré par un *contrôleur mémoire* (intégré au processeur ou au *chipset* du processeur).

Organisation d'un processeur Pentium



Bus système (*Front Side Bus*)
Remplacé par QPI (*Quick Path Interconnect)* puis par UPI (*Ultra Path Interconnect*) dans les versions récentes

NorthBridge
Assure les E/S rapides :

- mémoire (inclut le contrôleur mémoire)
- graphique (par AGP ou PCI Express)

SouthBridge
Assure les E/S plus lentes :
Comprend un contrôleur PCI (ou PCI Express)
des contrôleurs de périphériques

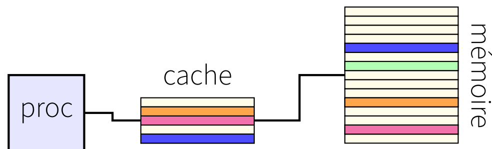
- SATA/IDE
- USB, Ethernet, ...

Se connecte via un bus LPC (*low pin count*) à des périphériques lents.

Les mémoires *cache*

un **cache** stocke une partie des données de la mémoire centrale

Organisé en *lignes de cache*



Lors d'un accès mémoire :

1. On cherche si la donnée est présente dans le cache
2. Si oui (succès ou *hit*), on envoie la donnée au processeur.
   - t_a **temps d'accès réussi**
3. Si non (échec ou *miss*)
   1. On lit la ligne contenant la donnée en mémoire et on la copie dans le cache.
   2. on envoie la donnée au processeur.

   - t_e **temps d'échec**
   - τ_e **probabilité d'échec**

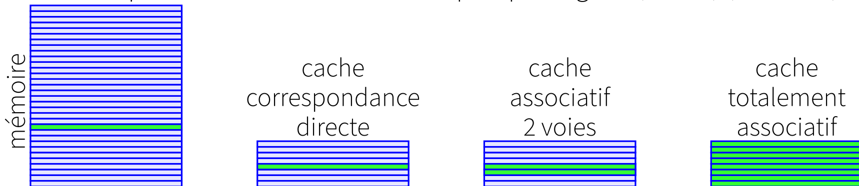
Où stocker une ligne dans le cache ?

Plusieurs politiques :

cache à correspondance directe (*direct map*) : ligne à un emplacement unique dépendant de son adresse.
Exemple : poids faibles de l'adresse → emplacement dans le cache
Problème : des lignes différentes avec les même poids faibles d'adresse sont dans la même ligne de cache

cache associatif : ligne à un emplacement quelconque
Évite que des lignes avec les mêmes poids faibles d'adresse ne s'*écrasent* mutuellement, mais nécessite une recherche complexe

cache associatif par ensemble (*set associative*) : ligne à un emplacement quelconque dans un ensemble (ou bloc) déterminé par l'adresse
Bon compromis, les ensembles font quelques lignes (4 .. 20) (ou voies)



Comment repérer si une ligne est présente dans le cache ?

<table>
<tr><td colspan="2">adresse de la ligne</td><td rowspan="2">adresse octet dans la ligne</td></tr>
<tr><td>étiquette</td><td>index</td></tr>
</table>

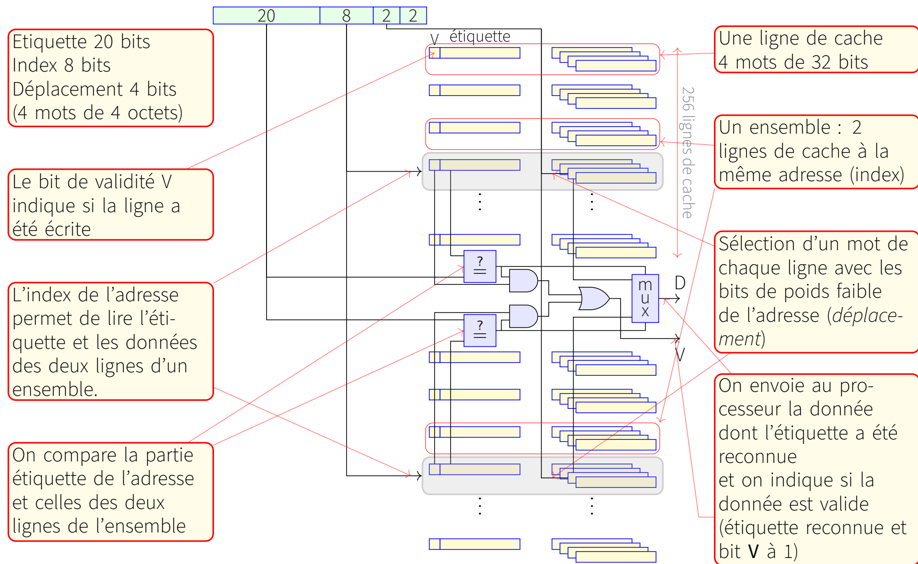
On distingue différent champs dans une adresse:

- l'adresse de l'octet dans la ligne : **déplacement** ou *offset*
- la position de la ligne dans le cache (= adresse de ligne pour les caches à correspondance directe ou adresse de bloc pour les caches associatifs par bloc) : **index**
- les poids forts identifient la ligne de cache : **étiquette** ou *tag*

Exemple : cache de 1024 blocs, 4 mots de 32 bits par ligne

- déplacement : adresse de l'octet dans le mot (2b) + adresse du mot dans la ligne (2b)
- index : 10 bits (repère un ensemble parmi 1024)
- étiquette : $32 - 10 - 4 = 18$ bits

Exemple de cache associatif 2 voies : 256 ensembles de 2 lignes de 16o



Comment remplacer une ligne ?

Si on insère une nouvelle ligne dans le cache, il faut en supprimer une.

Si le cache est (même partiellement) associatif, comment choisir la ligne à supprimer ?

Différentes politiques :

- aléatoirement
- la plus ancienne (FIFO ou *round-robin*)
- la moins fréquemment utilisée (*least frequentlu used* LFU)
- la moins récemment utilisée (*least recently used* LRU)

Les techniques LRU, LFU nécessitent de stocker beaucoup d'informations avec chaque ligne (nombre d'accès, date du dernier accès) et de les mettre à jour à chaque accès.
Il en existe des versions simplifiées (pseudo-LRU).

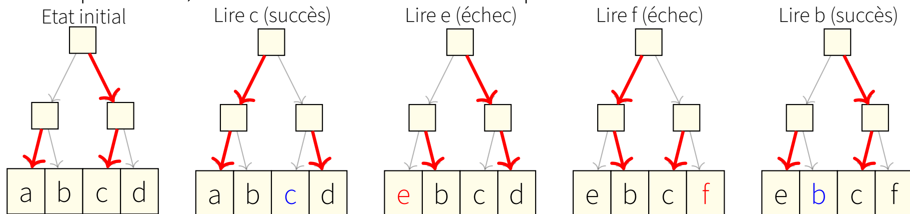
Pseudo LRU (PLRU)

1bit-PLRU

A chaque accès, indiquer avec un bit si la ligne est dans la partie haute ou basse de l'ensemble (bit de poids fort de son adresse dans l'ensemble).
En cas d'échec, prendre une ligne aléatoire de l'autre partie.

PLRU en arbre

On maintient dans un arbre binaire un chemin conduidant vers la ligne pseudo-LRU.
Nécessite $2^k - 1$ bits par ensemble pour un cache associatif 2^k–voies.
En cas d'échec, on suit le chemin vers la PLRU et on la remplace.
A chaque accès, on « écarte » de la cible chaque branche de l'arbre traversée.



Comment remplacer une ligne ? (cont.)

pseudo-LRU

bit-pLRU On associe à chaque ligne un bit d'état. A chaque accès à la ligne, le bit est mis à 1.
En cas d'échec on remplace une des lignes dont le bit d'état est à 0.

- S'il en a plusieurs, on en choisit une (aléatoirement, celle avec le plus petit index, ...) et on met son bit d'état à 1.
- S'il n'y a qu'une ligne dont le bit d'état est à 0, on la choisit et on complémente *tous* les bits d'état

NB : les différentes méthodes *pseudo LRU* ne garantissent *jamais* que c'est la ligne avec l'accès le plus ancien qui sera choisie.
Par contre, on est certain que ce ne sera pas la plus récente.

NB2 : LRU n'est pas forcément la meilleure politique. Si une ligne est accédée fréquemment, il est très possible qu'elle soit encore accédée, et il faut la préserver, même si c'est le (vrai) LRU.
Pour cette raison, certaines méthodes considérent différemment des lignes accédées une seule fois et des lignes accédées deux fois ou plus. En cas d'échec, on choisira plutôt les premières.

Cas des écritures

Deux politiques dans le cas d'une écriture :

Cache à écriture simultanée (*write-through*)

On écrit simultanément dans le cache et la mémoire pour assurer la cohérence des information.

Problème : encombrement du bus

Cohérence cache – mémoire au plus tôt

Cache à réécriture (*write-back*)

Lors d'une écriture en cache, on marque la ligne comme modifiée (***dirty bit***).

On réécrit la ligne en mémoire

- seulement quand elle est supprimée du cache
- si elle a été modifiée

Plus complexe, mais charge moins le bus.

Cohérence cache – mémoire au plus tard

Particulièrement critique dans les multicoeurs.

Cas des écritures (cont.)

Lors d'un défaut de cache en écriture (*cache write miss*), deux possibilités :

write allocate (ou *fetch-on-write*) :
: lors d'une écriture, on crée une entrée pour la ligne dans le cache. Nécessite une lecture de la ligne, mais plus efficace en cas d'écritures successives dans la même ligne

no-write allocate :
: la donnée est directement envoyée à la mémoire. On n'alloue une ligne que sur une lecture.

Généralement on combine :

- *write-back* et *write allocate*
- *write-through* et *no-write allocate*

Cache instruction et données

On peut avoir ou non les instructions et les données dans le même cache.

Caches séparés (*split caches*) :
: un cache pour les instructions et un pour les données.
Permet d'éviter les conflits d'accès à la mémoire dans le processeur (lecture simultanée cache instruction et cache donnée).

Cache unifié (*unified cache*) :
: un cache unique stockant instructions et données
Permet une meilleure utilisation de la capacité du cache.

Amélioration des caches

Idéalement, il faudrait réduire

taux d'échec τ_e
temps d'accès t_a
temps d'échec t_e

Conduit à des exigences la plupart du temps contradictoires.

Réduction du taux d'échec τ_e

Trois types d'échecs (les trois « C »)

Obligatoires (*Compulsory*) : taux d'échec lors de la **première utilisation** d'une donnée
arriveraient même avec un cache infini totalement associatif

Capacité taux d'échec du à une **capacité** trop faible
n'arriveraient pas avec un cache de capacité infinie
solution : augmenter la taille du cache

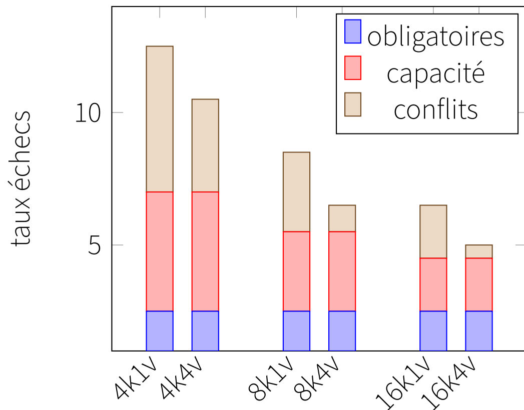
Conflits Taux d'échec du à des **conflits**
n'arriveraient pas avec un cache totalement associatif
solution : augmenter l'associativité

Amélioration des caches (cont.)

types d'échecs

Trois types d'échec : obligatoire (démarrage), capacité insuffisante, conflit (plusieurs adresses mémoire dans la même ligne de cache).

Taux d'échecs : caches de capacité 4k, 8k et 16k, associatifs 1 voies, 4voies



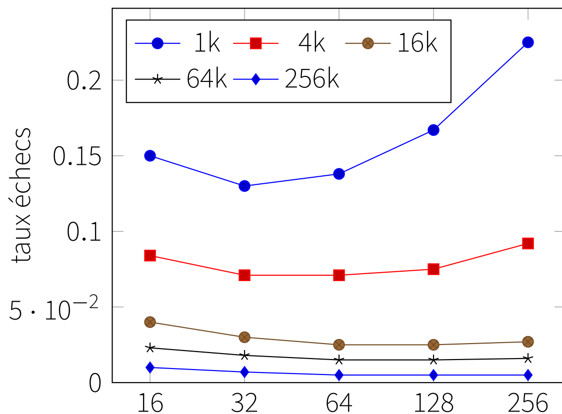
- Ni la capacité, ni l'associativité n'influent sur les échecs obligatoires.
- La capacité réduit les échecs de conflits et de capacité.
- L'associativé n'influence que les échecs dus à des conflits.

Réduction des échecs obligatoires (premier accès à une donnée).

- augmenter la taille des lignes : meilleure utilisation de la localité spatiale des données
- anticiper la demande : acquisition anticipée des données (*prefetch*)

Impact de la taille des lignes

Taux d'échecs en fonction de la taille des lignes pour différentes tailles de caches



Bon compromis : 32–64 o/ligne

Faible taille de ligne : échec car mauvaise utilisation de la localité spatiale

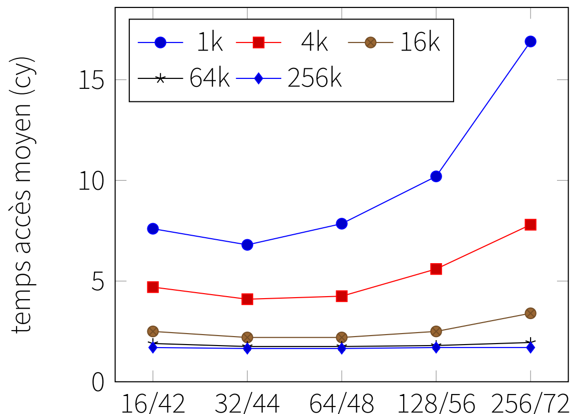
Grande taille de ligne : échec, car conflits plus importants (moins de lignes pour une capacité de cache donnée)

Amélioration des caches (cont.)

réduction des échecs obligatoires

En plus, augmenter la taille des lignes accroît la pénalité d'échec

Temps d'accès moyen en fonction de la taille des lignes pour différentes tailles de caches



Hypothèses : 40 cy puis 16o tous les 2 cy

Prend en compte le taux d'échec

ligne de faible taille : pénalité à cause du taux d'échecs obligatoires
ligne de grande taille : pénalité à cause du coût du transfert de la ligne et du taux d'échecs de conflits

Amélioration des caches (cont.)

réductions des échecs de conflits

Réduction des échecs dus à des conflits ($\tau_e \searrow$):

- augmenter l'associativité
  Cette augmentation est complexe pour de gros caches.
  - Organisation des caches en *bancs* de petites tailles.
    On cherche dans chaque banc, puis entre les résultats des bancs.
  - Prédiction de voie. On cherche d'abord dans la voie prédite dans un ensemble (précédemment utilisée), avant de faire une recherche associative sur toutes les voies de l'ensemble.
    La prédiction est mise à jour à chaque accès.
- Cache des victimes
  - Quand une ligne est éjectée, on la met dans un petit cache totalement associatif (typ. 8 entrées) (*victim cache*).
  - Si on a a nouveau besoin de cette ligne, elle sera directement disponible
  - Revient à augmenter l'associativité pour les ensembles dans lesquels il y a de nombreux échecs.

Amélioration des caches (cont.)

réduction temps échec

- Fournir les lectures mémoire (`ld`) au plus vite
  - Caches non bloquants :
    - exécution des requètes dans le désordre
    - *hit under miss*
    - priorité aux lectures sur les écritures
  - Délivrer au plus vite la donnée :
    - *early restart* : dès que le mot démandé est arrivé, l'envoyer au processeur sans attendre l'arrivée de la fin de la ligne
    - *critical word first* : la mémoire envoie en premier le mot critique
- Réduire le coût des écritures
  - Tampon des écritures (*write buffer*) : les données à écrire sont mises en attente dans un tampon sans attendre leur écriture effective
  - Fusion des écritures dans la même ligne de cache (pour caches à écriture simultanée (*write-through*))
- Caches à plusieurs niveaux. Deux, et même trois niveaux sont courants

Amélioration des caches (cont.)

réduction temps échec

On utilise *plusieurs niveau de caches* successifs pour permettre de respecter les exigences de réduction simultanée du temps d'accès t_a, du temps d'échec t_e et du taux d'échec τ_e.

le cache de niveau 1 (*level 1* ou L1) est le plus proche du processeur. Il doit fournir des données en 1 cycle et pour cela être le plus simple possible, quitte à avoir un taux d'échec un peu élevé.

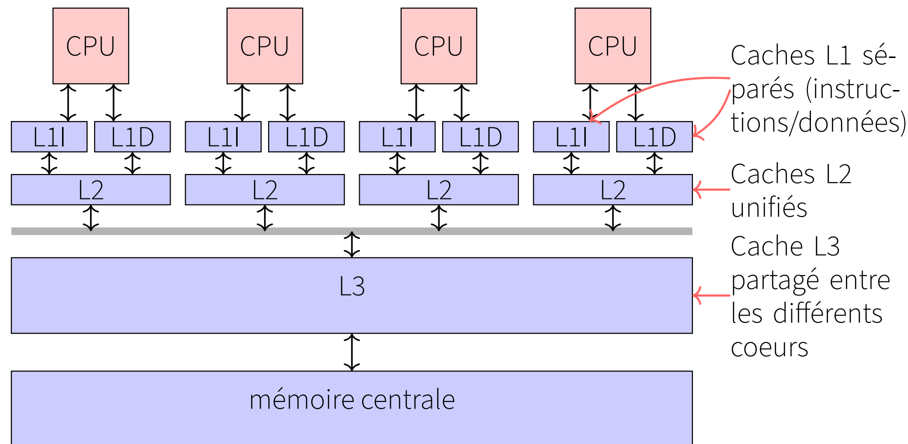
les caches de niveau supérieur (L2 et très souvent L3) doivent limiter les accès mémoire pour réduire le coût d'un échec. Il sont de plus grande taille et plus sophistiqués, mais leur temps d'accès est plus important (4–20 cy typ.) Le cache L2 est actuellement intégré au processeur, alors que L3 peut être intégré ou externe.

Caches à plusieurs niveaux

	Cache L1	Cache L2	Cache L3 (ou LL)
Objectifs	Cache rapide pour alimenter le processeur	Réduire le temps d'échec du cache L1 et limiter au maximum les accès à la mémoire	
Type de cache	Deux caches séparés données et instructions	Cache unifié instructions/données	
Temps d'accès	1–5 cy (latence, mais pipeliné: 1 accès/cy)	3–20 cy	10–40cy
Temps d'échec	3-15 cy	10-40 cy	200cy
Taille	2×16–32 ko	256 ko	1 Mo+
Associativité	Faible associativité (pour réduire t_a) 1–8 voies	Associativité plus importante 4–16 voies	
Écriture	Écriture simultanée (pour simplifier le cache) ou réécriture	Réécriture (pour limiter le trafic mémoire et la consommation)	

Caches à plusieurs niveaux (cont.)

Organisation des caches dans un circuit multicoeur



NB : les caches L1 et L2 des différents processeurs communiquent pour assurer la *cohérence mémoire*.

Caches à plusieurs niveaux (cont.)

Les caches successifs peuvent être *inclusifs* ou *exclusifs*

Caches strictement inclusifs : $L1 \subset L2 \subset MP$

Tout ce qui est dans L1 est aussi dans L2.
Simplifie la mise en œuvre

Caches non strictement inclusifs : $L1 \cap L2 \neq \emptyset$

L1 peut contenir une partie de L2 et réciproquement.

Caches exclusifs : $L1 \not\subset L2$ et $L1 \cap L2 = \emptyset$

Une ligne n'est jamais à la fois dans L1 et L2.
Évite de dupliquer l'information.

- échec de L1 et copie depuis la mémoire principale (MP) : on transfère le bloc MP $\rightarrow$ L1.
  Le bloc remplacé de L1 est mis dans L2
- échec de L1 et succès dans L2
  Le bloc de L2 est copié dans L1.
  Le bloc remplacé dans L1 est copié dans L2
- Un tampon permet de ne pas retarder les échanges L2–L1

Evaluation de performances

Il est nécessaire de prendre en compte les caches dans les évaluations de performances.

Nous supposons des caches avec un temps d'accès t_a, un taux d'échec τ_e. Le temps d'accès à la mémoire est t_M.

Temps d'accès moyen à la mémoire (TAMM (ou AMAT en anglais)) :

$$TAMM = t_{aL1} + \tau_{eL1} \times t_{L2}$$

Le temps d'accès apparent au cache L2 t_{L2} est :

$$t_{L2} = t_{aL2} + \tau_{eL2} \times t_M$$

d'où $TAMM = t_{aL1} + \tau_{eL1} \times (t_{aL2} + \tau_{eL2} \times t_M)$

Avec $t_{aL1} = 4$, $t_{aL2} = 20$, $\tau_{L1} = \tau_{L2} = 0.1$ et $t_M = 100$, cela donne $TAMM = 4 + 2 + 1 = 7$.

Evaluation de performances (cont.)

Temps d'exécution d'un programme sur un processeur pipeliné :
1/ Cas avec un cache L1 parfait ($\tau_{L1} = 0$) et de temps d'accès 1 cycle
$T_{ex} = n_{instr} \times T_c \times \overline{CPI}$
2/ Cas avec un cache L1 parfait de temps d'accès t_{aL1}.
impact pour les accès mémoire données (`ld` et `st`),
mais pas pour les instructions car cache et processeur sont pipelinés.
$T_{ex} = n_{instr} \times T_c \times (\overline{CPI} + MPI \times t'_{aL1})$
où MPI = accès mémoire par instruction et $t'_{aL1} = t_{aL1} - 1$
3/ Cas général avec cache L1 imparfait de taux d'échec τ_{L1D} et τ_{L1I} (pour instructions et données)

$$\begin{aligned} T_{ex} = \; & n_{instr} \times T_C \times (\overline{CPI} \\ & + MPI \times (t'_{aL1} + \tau_{L1D} \times t_{L2}) \quad \text{(ld et st)} \\ & + \tau_{L1I} \times t_{L2}) \quad \text{(lecture instruction)} \end{aligned}$$

Soit $T_{ex} = n_{instr} \times T_C \times (\overline{CPI} + (TAMM - t_{aL1}) + MPI \times (TAMM - 1))$
Avec les hypothèse précédentes et $MPI = 0.2$, cela donnerait
$T_{ex} = n_{instr} \times T_C \times (\overline{CPI} + (7 - 4) + 0.2 \times 6) = n_{instr} \times T_C \times (\overline{CPI} + 4.2)$

Evaluation de performances (cont.)

Sur un processeur superscalaire, une évaluation analytique est *beaucoup* plus complexe et, en pratique, quasiment impossible.

Le parallélisme d'instructions (notamment pour un processeur superscalaire exécutant les instructions *dans le désordre*), permet de *masquer la latence des accès mémoires* en exécutant d'autres instructions pendant ce temps.

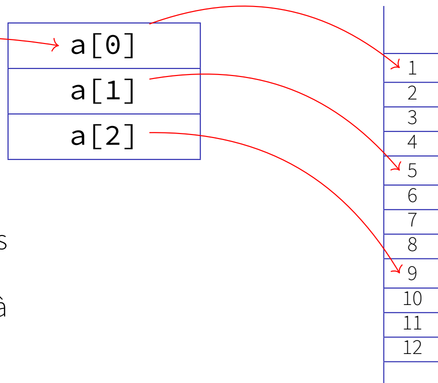
Le parallélisme de *threads* (*temporal multithreading*), permet également de masquer le temps des accès mémoire.

- duplication des registres architecturaux et du `pc` dans le processeur pour gérer deux *threads*.
- lors d'un échec du cache, on bascule instantanément sur le deuxième *thread*

Cache et optimisation de programmes

Les tableaux 2D en C sont alloués dans l'ordre « ligne d'abord » (*row major order*)[4].
Données d'une ligne dans des cases mémoires contiguës

```
int a[3][4]={
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
};
```



`a` est un tableau de pointeurs vers des lignes (tableaux 1D).
`a[i][j]` est équivalent à `*(*(a+i)+j)`.

[4] En fortran (et en matlab), c'est l'inverse. Les éléments sont rangés par colonne (*column major order*)

Cache et optimisation de programmes (cont.)

Le balayage à travers les colonnes dans une même ligne accède aux éléments successifs et permet d'exploiter la localité spatiale :

```
float a[N][N], sum;
for (i = 0; i < N; i++)
  sum += a[0][i];
```

Le premier accès(`a[0][0]`) sera un échec (obligatoire).
Si le cache peut contenir k floats, les $k - 1$ accès suivants seront des succès.
Taux d'échecs obligatoires $1/k = 4/B$ (si B est la taille d'une ligne de cache en octets)

Le balayage à travers les lignes dans une même colonne accède à des éléments distants

```
for (i = 0; i < N; i++)
  sum += a[i][0];
```

Aucune localité spatiale ! Tous les accès sont des échecs.
Taux d'échecs obligatoires = 100%

Cache et optimisation de programmes (cont.)

- L'accès aux éléments d'un tableau avec un pas 1 est favorable (localité spatiale)
- Un accès répété aux mêmes données est favorable (localité temporelle)

Exemple : somme des éléments d'une matrice d'entiers (ligne cache = 64 octets)

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

Taux d'échec = 6.25% (si ligne = 16 `ints`)

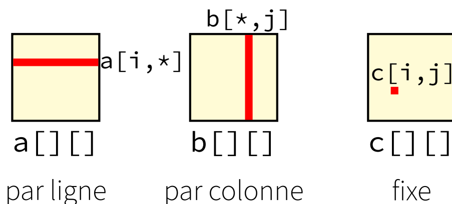
```
int sumarraycols(int a[M][N])
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
      sum += a[i][j];
  return sum;
}
```

Taux d'échec = 100%

Multiplication de matrices

Multiplication de matrices : $C = A \times B$ avec $C_{ij} = \sum_k A_{ik} \times B_{kj}$
Version : suite de produits scalaires (*ijk*)

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

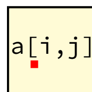


Echecs pour chaque itération de la boucle interne (ligne=8 doubles):
a : 0.125   b :1.0   c: 0
1.125 N^3

Multiplication de matrices (cont.)

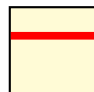
Réalisation sous forme SAXPY/DAXPY (*ikj*)

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



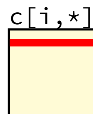
a[][]

fixé



b[][]

par ligne



c[][]

par ligne

Echecs pour chaque itération de la boucle interne (ligne=8 doubles):

a : 0.0 b :0.125 c: 0.125

0.25 N^3

Multiplication de matrices (cont.)

Autre solution : transposer préalablement la deuxième matrice.

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

La transposition introduit des défauts de cache, mais seulement N^2 accès (au lieu de N^3 pour la multiplication de matrices).
La multiplication se fait sans défaut de cache (autre qu'obligatoires)

Multiplication de matrices (cont.)

Multiplication par blocs.

On décompose la matrice en sous-matrices.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

avec $C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$, etc

Peut se généraliser pour un nombre de blocs quelconques

Si les blocs sont de taille suffisamment petite pour tenir entièrement dans le cache, il n'y aura plus d'échec (autre qu'obligatoire) quelle que soit la méthode.

Préacquisition de données

Une manière de réduire les défauts de caches obligatoires et leur latence est d'acquérir *en avance* les données ou les instructions. ***prefetch***
Ceci peut être fait par le matériel, le compilateur ou le programmeur.
Quelle ligne, quand, où, comment ?
Quelle ligne acquérir ? Une préacquisation incorrecte consomme bande passante, espace dans le cache, etc

- la suivante
- utilisation de *prédiction* par le programmeur (qui sait de quelles données il va avoir besoin), le compilateur (qui connait les structures de données) ou le matériel (en observant les défauts de cache).

Quand faire un *prefetch* ?

- trop tôt : risque d'éviction du cache
- trop tard : gain faible
- de plus, charge du bus par le *prefetch* et risques de dégradation des performances du cache pour ses accès « normaux »

Ou faire le *prefetch* ?

- dans le cache, mais risque d'éviction de lignes utiles (pollution du cache)
- dans un tampon spécial (*prefetch buffer*)

Préacquisition de données (cont.)

Préacquisition matérielle :

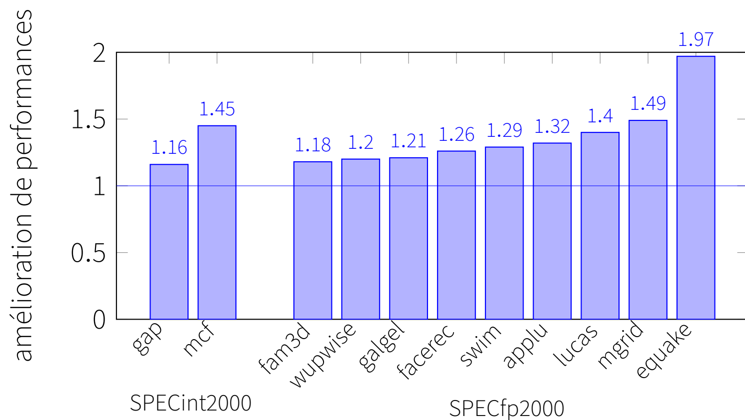
Instructions Deux mécanismes parallèles de *prefetch* (dans `L1I`) :
- linéaire (dans l'ordre des `pc`)
- avec utilisation des prédictions du `BTB`

Données Très compliqué à mettre en oeuvre
- Acquisition de la ligne suivante en cas d'échec sur une ligne (simple mais parfois inefficace)
- Utilisation du PC de l'instruction `ld` et recherche d'une régularité dans les pas d'accès (*stride*).
- *Stream buffers* : un SB est un tampon de profondeur k (typ.4) contenant des lignes préaquises $A+1, \cdots, A+k$
  Si échec à l'adresse A, chercher si un SB a les données à cette adresse
  - Si oui, préacquisition de la ligne $k+1$
  - Si non, allocation d'un nouveau SB pour l'adresse A et *prefetch*

  Mécanisme étendu pour gérer des pas différents de 1 (*stride*), des accès rétrogrades, etc par comparaison en parallèle des adresses dans les SB.

Préacquisition de données (cont.)

Améliorations des performances dues à une préacquisition matérielle (Pentium 4)



Préacquisition logicielle

Existence d'instructions de préacquisition générées par le compilateur (ou le programmeur).

Sur les Pentium, il existe 4 instructions de *prefetch*

`prefetch0` acquisition en L1

`prefetch1` acquisition en L2

`prefetch2` acquisition en L3

`prefetchnta` préchargement dans un cache particulier *non temporel*. Les données préacquises ne sont pas conservées (sauf si elles ont utilisées).[5]

---

[5] Dans les pentium récents, `prefetchnta` mets les données dans le cache L1, mais sans copie dans L2 et L3. Ainsi, pas d'échange de données de cohérence avec les autres processeurs dans un contexte multiprocesseurs.

Conclusion sur les caches

En pratique la combinaison de ces différentes techniques permet de réduire très fortement le nombre d'accès mémoire.

Défauts de cache L1I : 1–3%

Défauts de cache L1D : 4–10%

Défauts de cache L2/ts les accès cache : 1–3%

Défauts de cache L3/ts les accès cache : 0,1–1%

Pénalité moyenne due aux accès mémoire (TAMM-1) : 0,1–1

Dépend fortement du programme, du compilateur et ... du programmeur.

Mémoire virtuelle

Systèmes modernes : extension de la mémoire centrale en utilisant une partie de la mémoire de masse (disque) → « **mémoire virtuelle** »
En fonction des besoins, les informations sont transférées entre disque et mémoire centrale par *pages* (typ. 1-4 ko).

Mémoire centrale ≡ *cache* pour la mémoire virtuelle

temps d'accès disque très important ≈ 1 ms (10^6 cy)

⇒ ***associativité totale*** pour limiter le taux d'échec

⇒ programme et données peuvent être à n'importe quel emplacement en mémoire centrale suivant la place disponible.

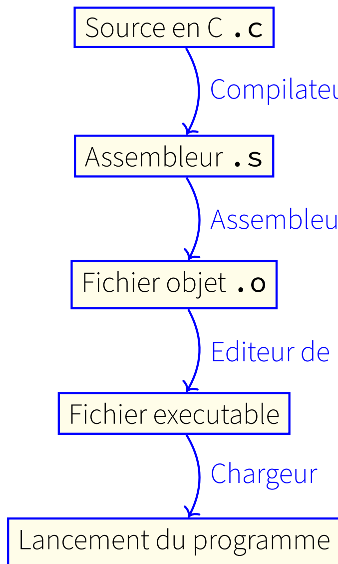
Mais les adresses vues par le processeur *doivent* rester identiques.

adresses virtuelles vues par le processeur
fixées à la compilation, caractéristiques du programme

adresses physiques en mémoire centrale
liées à une exécution du programme, varient dynamiquement

Mémoire virtuelle (cont.)

rappels sur processus de développement logiciel

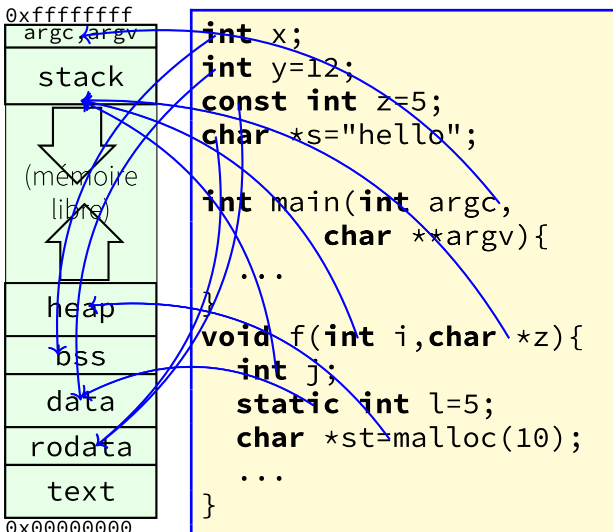


compilateur : traduit en assembleur un fichier C.

assembleur : convertit le programme en fichier objet.
Il contient le code binaire des instructions et des symboles donnant les noms des variables, fonctions, etc.

éditeur de liens (ou *linker*) : rassemble les fichiers objets et les bibliothèques.
Il positionne les variables globales et statiques en mémoire, ainsi que les fonctions, et modifie le code pour générer les adresses correspondantes.

chargeur (ou *loader*) : demande des ressources au système d'exploitation.
Il positionne divers registres et tables et lance l'exécution du programme.



Segments mémoire
(présents dans l'exécutable et créés par le *loader*)

- **pile** (`stack`) gestion des appels de fonctions
- **tas** (`heap`) allocations dynamiques (`malloc()`)
- **bss** données non initialisées
- **data** données initialisées
- **rodata** données constantes
- **text** ou (code) instructions du programme

Mémoire virtuelle (cont.)

rappels : segments mémoire d'un exécutable

Un compilateur va toujours générer les informations suivant ce shéma, avec la première instruction du programme à l'adresse 0.

Par contre, à l'exécution, le programme et ses données seront placés en mémoire en fonction de l'espace libre à un moment donné.

Lors du lancement du programme, le *loader* va (entre autres) :

- lire l'entête du fichiet objet, pour connaitre la taille des différents segments.
- demander au système de créer un *processus* et d'allouer de la place en mémoire centrale pour les différents segments.
  - Le système associe au processus une table permettant de calculer les adresses en mémoire (*adresses physiques*) en fonction des adresses manipulées par le processeur (*adresses virtuelles*) (***table des pages***). Certains segments [*text*,*rodata*] sont *read-only* et partagés entre les différentes évocations du programme.
- charger les instructions dans le segment *text*, les données dans les segments *data* et *rodata*, copier les arguments du programme en haut de la mémoire, mettre à zéro le segment *bss* et lancer le *main*.

Mémoire virtuelle (cont.)

Séparation adresses virtuelles (vues par le processeur) / adresses physiques (mémoire)

$\Longrightarrow$ permet l'exécution de plusieurs tâches par un processeur.

Nécessaire (même en l'absence de mémoire de masse) pour avoir un système d'exploitation[6] mutitâches.

Avantages :

- les processus voient toujours une mémoire contiguë et identique[7], alors que les zones correspondantes en mémoire physique peuvent être disjointes
- chaque process a son propre *espace d'adressage* auxquels les autres process ne peuvent pas accéder (sauf cas particulier de segments partagés (*text,rodata*), mécanisme de communication entre process (*IPC*), etc)

[6] La plupart des applications embarquées s'exécutent dans un système d'exploitation. Mais certaines sont sans système et donc sans nécessité de gestion mémoire (*bare-metal)*.

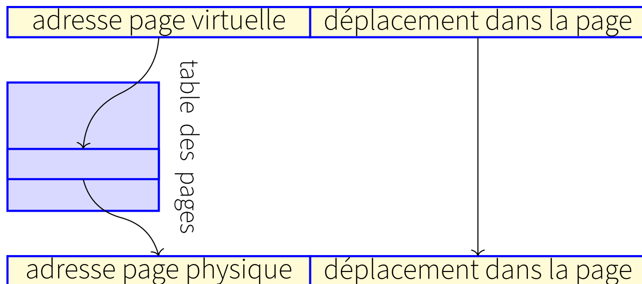
[7] Pour des raisons de sécurité, les emplacement des principaux segments peuvent être positionnés aléatoirement au lancement du programme (*address space layout randomization*).

Mémoire virtuelle (cont.)

traduction adresses virtuelles → physiques

Pour gérer la mémoire virtuelle, structuration en *pages*.

Le système d'exploitation gère par logiciel une *table des pages* donnant pour chaque page la correspondance entre l'adresse virtuelle (du programme) et l'adresse physique (en mémoire), ainsi que les droits d'accès (RO/RW, partagé/privé, instruction/données, etc).



Mécanisme de traduction adresses virtuelles → adresses physiques

Mémoire virtuelle (cont.)

tables des pages

Ces tables sont de *grande* taille et sont elles même en mémoire.

Exemple :

Adresse mémoire sur 32 bits, pages de 4ko (valeurs typiques 1–8ko).

Nombre de pages : $2^{32}/2^{12} = 2^{20}$ =1Mpages

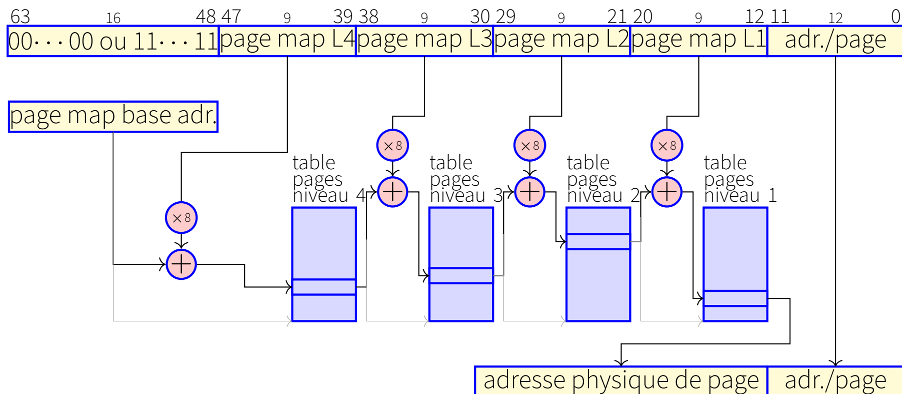
4Mo de table pour décrire la table des pages pour **chaque** process.

Pour les systèmes avec une largeur d'adresse plus importante (jusqu'à 64 bits d'adresses actuellement), la taille des tables devient démesurée ($2^{64}/2^{12}$!!) alors que la plupart des entrées sont inutilisées :

Une organisation en tables sur plusieurs niveaux permet de réduire **très fortement** cette taille.
A chaque niveau, seules les tables utiles sont créées.

Exemple d'organisation du calcul des pages virtuelles en 4 niveaux (x86/IA-64)

Pages 4ko (2^{12}), adresses virtuelles 64bits (8o), adresses physiques 48 bits.
$\implies$ 2^9 adresses physiques de 2^3 octets dans une page.
Entrée de niveau i = adresse de la table $i-1$. Il faut ajouter le déplacement dans champ $i-1$ de l'adresse virtuelle ($\times 8$ car adresses sur 8 octets).



Pour faire un accès mémoire, le processeur doit donc :

- pour chacun des niveaux de table de pages
  - extraire le champ d'adresse correspondant et l'ajouter à l'adresse de base de table des pages
  - lire la mémoire pour aller chercher dans la table des pages à cette adresse, soit la nouvelle adresse de table de page, soit l'adresse physique de la page
- calculer l'adresse réelle (concaténation des poids faibles de l'adresse (virtuelle) et de l'adresse de page physique)
- faire l'accès mémoire

« ***page walk*** »

Mémoire virtuelle (cont.)

TLB

Nécessité de faire 1 à 4 accès mémoire supplémentaires pour **chaque** lecture/écriture mémoire

Ce mécanisme est évidemment beaucoup trop lourd.

$\Longrightarrow$ cache particulier **TLB** (***translation lookaside buffer***)

- dans le processeur
- mémorise quelques dizaines de correspondances entre adresse de page virtuelle / adresse de page physique.

Pour chaque accès mémoire, on lit le TLB pour avoir l'adresse de page physique correspondant à une adresse virtuelle

En cas d'échec du TLB(*TLB miss*), on va lire la table des pages (*page walk*), et on recharge la valeur.

Le cache peut être soit dans le domaine virtuel, soit dans le domaine physique.

Cache virtuel : Le cache traite directement les adresses (virtuelles) du processeur.
Le TLB est *après* le cache (ce qui simplifie).
Par contre, quand on change de process ou que le SE reprend la main, risque d'avoir :

- deux adresses virtuelles identiques dans deux process avec des adresses physiques ≠ (*homonymes*)
- deux adresses virtuelles ≠ de deux process correspondant à la même adresse physique (*synonymes*)

Nécessité de vider le cache en cas de changement de process (ou même pour traiter un appel système).

Cache physique : traduction virtuel/physique *avant* le cache.
Le TLB est un organe **très** critique pour les performances.
Mais, pas de risque de confusion d'adresse.

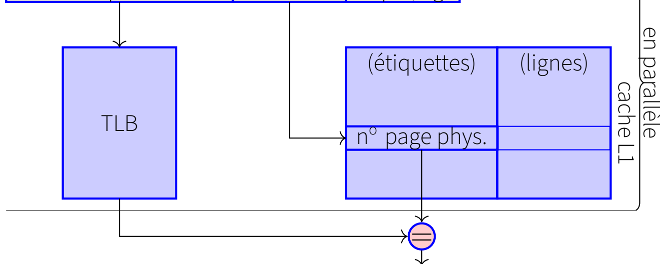
Mémoire virtuelle (cont.)

index virtuel, étiquette physique

Bon compromis : **caches à index virtuel et étiquette physique**
Permettent de faire recherche dans cache et traduction TLB en parallèle.

Il *faut* pour cela taille(index cache + déplt./ligne) $\leq$ taille(déplt./page)

adresse virtuelle	n° de page virtuelle	déplt. / page	
adresse dans le cache	étiquette	index	déplt. / ligne



Si le nombre de bits idx+depl.==depl./page l'étiquette est le numéro de page physique.

Mémoire virtuelle (cont.)

Sur la plupart des architectures modernes, les caches sont à index virtuel et adressage physique.
Cela résoud les problèmes d'homonymie/synonymie, et reste matériellement réalisable.

Sur certaines architectures plus anciennes (MIPS R4000, arm7, arm9, arm11), les caches L1 sont entièrement virtuels pour simplifier le matériel et déporter le TLB après le cache L1.
Dans ce cas, pour résoudre le problème des homonymes lors d'un changement de process, on ajoute à l'étiquette un identificateur d'espace d'adressage associé au process par le SE (ASID).
Des process $\neq$ ont des ASID $\neq$ et des adresses identiques avec des ASID différents feront un défaut de cache.

Par contre, pas suppression des synonymes.
Peut être résolu en supprimant du cache les adresses correspondant à des segments mémoire partagés lors d'un changement de process.

Le TLB fonctionne comme un cache donnant l'adresse de page physique.

cache :

adresse de ligne	déplt. / ligne

- utilisé pour repérer la ligne de cache donnant le contenu d'une ligne mémoire
- découpé en index et étiquette

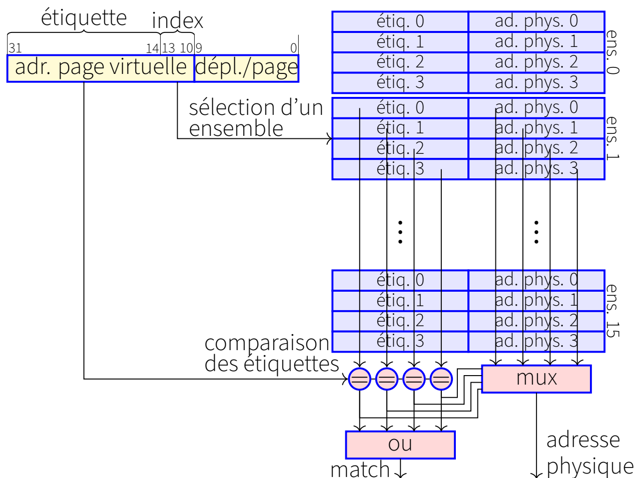
TLB :

adresse de page	déplt. / page

- utilisé pour repérer l'entrée de TLB donnant l'adresse physique d'une page virtuelle
- découpage en index et étiquette

Comme tout cache, le TLB peut être à correspondance directe ou associatif.

Exemple : TLB de 64 entrées associatif 4 voies (16 (2^4) ens. de 4 entrées, 4 bits d'*index*).
Adresses sur 32 bits, pages de 1ko (2^{10} o, 10 bits déplt./page).



Déplacement par page sur 10b

Adresse de page virtuelle (32—10b) décomposée en

- index repérant un ensemble sur 16 (4b)
- étiquette (32—4—10b) identifiant la page

Mémoire virtuelle (cont.)

protection mémoire

La table des pages comprend également des informations pour la gestion et la protection de la mémoire.
Un accès non autorisé créera une exception : *défaut de segmentation*

Les informations peuvent être :

- lecture (droit de lire la page)
- écriture (droit d'écriture)
- exécution (la page contient des instructions — implique généralement non écriture)
- partagée (page partagée entre plusieurs process)
- disque (la page est copiée sur le disque (*swap*))

En cas d'accès incorrect, une exception gérera la problème : suivant le cas : arrêt du programme (*segmentation fault*), transfert de la page du disque vers la mémoire centrale (*page fault*).

Il y a également diverses informations associées aux pages pour le remplacement de pages (LRU) en cas de défaut de page.
Ces informations sont présentes dans le TLB et l'exception éventuellement créée lors de la traduction physique-virtuel.

Mémoire virtuelle (cont.)

La plupart du temps, le cache L1 est pipeliné et accédé en 3–5 cycles.
Les 1er cycles permettent de faire la traduction virtuel-physique avec le TLB.

Il y a généralement un TLB indépendant pour instruction et données (pour éviter les aléas structurels).

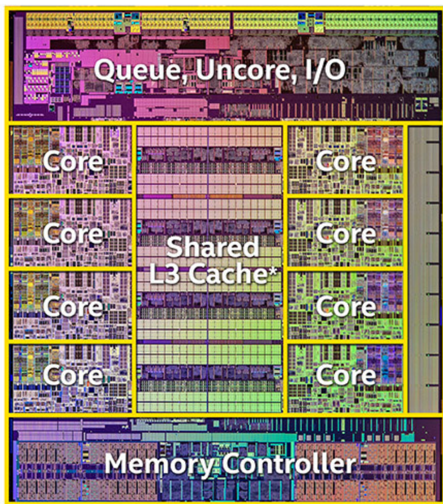
Il y a le plus souvent un TLB unifié de 2ème niveau. Les TLB-L1I et TLB-L1D comprennent quelques dizaines entrées, le TLB-L2 quelques centaines.
Comme tout cache, les TLB peuvent être ou non associatifs.

En cas d'échec (*TLB miss*), il faut calculer l'adresse de page physique en parcourant les différentes tables de pages (*page walk*). Peut se faire :

- par génération d'une exception et exécution d'une routine logicielle
- par une gestion matérielle

Récapitulatif

Exemple du processeur core i7-6900 (8 coeurs) (μarchitecture Nehalem)



Hiérarchie mémoire du i7-6900

Caches (strictement inclusifs)

	taille	caractéritiques	latence
L1 D cache	32 Ko	64o/ligne, Ass. 8 voies	4-5 cy (pipeliné)
L1 I cache	32 Ko	64o/ligne, Ass. 4 voies	3 cy (pipeliné)
L2 cache (unifié)	256 Ko	64o/ligne, Ass. 8 voies	12 cy
L3 cache (partagé)	8 Mo	64o/ligne, Ass. 12 voies	36 cy
RAM			36 cy + 57 ns

TLB (pages 4Ko) (également configurable avec des pages de 2 ou 4Mo)

	taille	caractéritiques	latence
TLB L1 (D)	64 entrées	Associatif 4 voies	Hit : 1 cy, miss 8 cy
TLB L1 (I)	128 entrées	Associatif 4 voies	Hit : 1 cy, miss 8 cy
TLB L2 (I et D)	1024 entrées	Associatif 8 voies	Hit : 8 cy, miss 20–800 cy[8]

[8] Suivant que les entrées utiles successives de la table des pages sont en mémoire centrale ou dans un des caches.

Récapitulatif (cont.)

Hiérarchie mémoire Arm-Cortex A-9

Caches (exclusifs)

	taille	caractéritiques	latence
L1 D cache	16–32 Ko	64o/ligne, Ass. 4 voies	4 cy (pipeliné)
L1 I cache	16–32 Ko	64o/ligne, Ass. 4 voies	3 cy (pipeliné) (virtuel)
L2 cache (unifié)	256k–1Mo	64o/ligne, Ass. 4 voies	19 cy
L3 cache (partagé)	0–4 Mo	64o/ligne, Ass. 8 voies	?? cy (opt.)
RAM			110 ns

TLB (pages 4Ko)

	taille	caractéritiques	latence
TLB L1 (D)	32 entrées	Associatif complet	Hit : 1 cy, miss 7 cy
TLB L1 (I)	32 entrées	Associatif complet	Hit : 1 cy, miss 7 cy
TLB L2 (I et D)	128 entrées	Associatif 2 voies	Hit : 7 cy, miss ??? cy