Systèmes multicoeurs : aspects architecturaux

Alain Mérigot

Université Paris Saclay

Table of content

1 Modéle de mémoire partagée

2 Instructions de synchronisation

3 Cohérence de cache
- Position du problème

4 Protocoles de cohérence
- Protocoles à mise à jour et protocoles à invalidation
- Protocole MSI
- Protocole MESI
- Protocoles à répertoire

Cohérence et consistence de la mémoire dans un multiprocesseur

On suppose un multicoeur à mémoire partagée et les programmes suivant s'exécutant sur deux processeurs.

`a` et `flag` sont des variables partagées en mémoire
Initialement `a=0` et `flag=0`

Thread A

```
a=1;
flag=1
```

Thread B

```
while (flag==0)
  /* attendre flag !=0 */ ;
printf("a=%d\n",a);
```

On s'attend à avoir l'affichage `a=1`...

Mais cela dépend du comportement des ordres de lecture et d'écriture de la mémoire.

Cohérence et consistence de la mémoire dans un multiprocesseur (cont.)

Problèmes :

Rien ne garantit :

- l'ordre des écritures de `a` et `flag` par le processeur 1
- l'ordre d'exécution des instructions de lecture du processeur 2

Cet ordre peut être changé :

- par le compilateur (aucune dépendance entre les instructions)
- par le processeur (exécution *out-of-order*)
- par un cache à réécriture (*write-back*)
- par le tampon d'écriture (*load-store-queue*)

Ceci pose un gros problème pour l'écriture de programmes parallèles.
Nécessité de préciser le comportement vis à vis des accès mémoire

Cohérence et consistence de la mémoire dans un multiprocesseur (cont.)

Deux notions importantes :

Cohérence mémoire :
assure que les valeurs des différents caches sont identiques
La vision de la mémoire est cohérente entre les différents processeurs

Consistence mémoire :
concerne l'ordre dans lequel les changements de la mémoire sont vus par les différents processeurs

Cohérence et consistence de la mémoire dans un multiprocesseur (cont.)

Cohérence mémoire

Assuré par la plupart des multicoeurs actuels par des mécanismes matériels décrits plus loin

Avantages :

- permet de simplifier la programmation parallèle.
- Tous les processeurs voient les mêmes valeurs.

Inconvénients :

- Complexe à mettre à oeuvre
- Ralentit les accès mémoire
- Rend les temps d'accès à la mémoire non prédictibles (problématique pour le temps réel)

Consistence mémoire

Concerne l'ordre des lectures et écritures en mémoire

Il existe différents modèles de consistence mémoire

Terminologie :

Ordre du programme : ordre dans lequel un programme émet des lectures ou écritures

- ne concerne qu'un processeur (local)
- peut varier d'une exécution à l'autre, même sur un un monocoeur (caches *write-back*, exécution OoO, spéculation, LSQ, etc)

Ordre de visibilité : ordre dans lequel les lectures et écritures sont vues par différents processeurs

- concerne tous les processeurs d'un multicoeur
- peut être différent suivant les processeurs
- chaque processeur lit la dernière valeur en mémoire dans l'ordre de visibilité

Cohérence et consistance de la mémoire dans un multiprocesseur (cont.)

Modèles de consistance mémoire

Des modèles théoriques plus ou moins contraignants peuvent être définis.

Consistance stricte : toute écriture par un processeur est vue *instantanément* par *tous* les processeurs.
Modèle très fort, impossible à réaliser

Consistance séquentielle (Leslie Lamport) : les écritures des variables doivent être vues dans le même ordre par tous les processeurs.
Les processeurs doivent modifier la mémoire dans l'ordre du programme.
Facile à utiliser dans des programmes et à analyser, mais difficile à mettre en oeuvre efficacement :

- pas de réordonnancement des lectures écritures (compilateur, processeur)
- pas de cache à réécriture
- pas d'écriture combinée d'une ligne de cache dans le LSQ

Cohérence et consistence de la mémoire dans un multiprocesseur (cont.)

Consistence causale : seules les opérations ayant une causalité (dépendance) ont besoin d'être vues dans le même ordre par les différents processeurs.
Difficile à analyser (par exemple à cause des alias mémoire).

Consistence de cache : toutes les opérations d'écritures à une même adresse mémoire sont vues dans le même ordre par tous les processeurs

Consistence faible : seules *certaines* opérations d'accès à la mémoire nécessitent un ordonnancement. Les autres opérations peuvent être dans un ordre quelconque.
Permet d'assurer une consistence mémoire pour des opérations critiques (synchronisations, sémaphores, données partagées entre *threads*, etc) identifiées par le programmeur.
On utilise généralement ce modèle.

Table of content

1 Modéle de mémoire partagée

2 Instructions de synchronisation

3 Cohérence de cache
- Position du problème

4 Protocoles de cohérence
- Protocoles à mise à jour et protocoles à invalidation
- Protocole MSI
- Protocole MESI
- Protocoles à répertoire

Support matériel des synchronisations et de l'exclusion mutuelle

La mise en oeuvre correcte des synchronisations suppose un support matériel.
Haie (*fence*) Pour assurer un séquencement correct, permet d'interdire, lors du réordonnancement des instructions par un processeur, de dépasser une haie.
Toutes les instructions modifiant la mémoire avant la haie devront être exécutées avant l'exécution de l'instruction de haie.
Peut être au niveau du compilateur (*software fence*), du processeur (*harware fence*), ou les deux.
Une haie logicielle interdit au compilateur tout réordonnancement.
Exemple gcc : `__asm__ __volatile__ ("" ::: "memory");`

Le compilateur ne déplacera aucune instruction d'accès à la mémoire à travers la haie.

Une haie matérielle est une instuction exécutée par le processeur qui interdit le déplacement d'instructions mémoire par le processeur

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Sur le pentium instruction `mfence`
L'instruction `mfence` interdit que des `ld` ou `st` *après* le `mfence` s'exécutent tant que *tous* les accès mémoire en attente ne sont pas terminés et la LSQ vidée.

Il existe de manière similaire :
`lfence` idem uniquement pour les `ld`
`sfence` idem uniquement pour les `st`

Sur le processeur Arm, instruction **dmb** (*data memory barrier*) au comportement similaire (non réordonnancement des accès mémoire)
L'instruction `dsb` (*data synchronization barrier*) interdit *tout* réordonnancement d'instructions (accès mémoire ou non).

Instructions atomiques

Une instruction *atomique*[1] réalise un cycle lecture-modification-écriture (*read-modify-write*) sans interruption possible.

Ces instructions sont indispensables dans un système d'exploitation pour pouvoir mettre à jour des variables du noyau de manière sure (par un seul processus).

Elles permettent également des modifications en mémoire par un seul *thread*, sans perburbation par les autres *threads*
qu'il s'agisse d'autres threads du même processeur
ou de *threads* s'exécutant sur un autre coeur.

[1] Au sens étymologique : incassable

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Test and Set (TAS)

- Lit une case mémoire dans un registre
- Met la case mémoire à 1

Doit être *atomique*. Aucun autre *thread* ou process ne doit pouvoir accéder à la variable pendant cette instruction.
Permet d'acquérir un *mutex* par attente active (*spinlock*)

```
void acquerir(int * verrou){
  while( TAS(verrou) == 1) {
    ;
  }
}
```

Pour le libérer, pas de problème particuler car le *thread* est propriétaire du verrou :

```
void relacher(int * verrou){
  *verrou = 0;
}
```

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Il n'y a pas de réordonnancement d'instructions par le processeur quand il y a une instruction TAS.
Pour le compilateur, le TAS doit être protégé par une haie pour éviter un réordonnancement.

Peut être très coûteux à mettre en oeuvre :

- Dans un contexte multiprocesseur, le bus mémoire doit être verrouillé pendant tout l'accès mémoire.
- Le processeur doit faire une lecture et une écriture *vers la mémoire centrale*.
- Gestion de la boucle (sans spéculation).

Sur le Pentium, se fait avec une instruction (atomique) `xchg`

```
1 mov  eax, 1          ; Mettre le registre eax a 1
2 xchg eax, [verrou]   ; Echange atomique entre eax et
3                      ; la case d'adresse memoire verrou
```

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Autres instructions atomiques pour l'exclusion mutuelle

Fetch-and-Add (`faa`)

- Ajoute une valeur à la mémoire
- lit la valeur initiale dans un registre

Permet de mettre en oeuvre un verrou par *ticket d'attente*.

```
struct verrou {
  int numeroattente;
  int tour;
}
void lock(verrou *v){
  int montour = FetchAndAdd(v->numeroattente, 1);
  while(v->tour != montour) ;
}
```

Le *spinlock* de verrouillage n'utilise pas d'instruction atomique, mais une simple lecture.

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Compare-and-Swap(`emplacement, ancien, nouveau`) (`cas`)

- lire `emplacement` dans `valeur`
- Si `valeur == ancien`, alors charger `nouveau` dans `emplacement`
- renvoyer `valeur`

Permet de mettre en oeuvre des algorithmes de synchronisation non bloquants[2] (*lock-free*).

`cas` est plus puissant que les autres instructions de synchronisation.

[2] C'est à dire qu'une défaillance d'un *thread* ne perturbe pas les autres *threads*

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Ces mécanismes de modification atomique d'une donnée en mémoire existent dans la plupart des processeurs.

Exemple: **x86**

`xchg`	échange atomique mémoire registre. Equivalent TAS
`lock`	exécute l'instruction suivante avec le bus verrouillé
`lock xadd`	*fetch and add* atomique
`cmpxchg`	*compare-and-swap* 32 bits
`cmpxchg8b`	*compare-and-swap* 64 bits
`cmpxchg16b`	*compare-and-swap* 128 bits

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Dans les processeurs RISC, pas d'instructions complexes.
L'alternative consiste à utiliser des accès mémoire *verrouillés*.
`ll` *Load Locked*[3]
`sc` *Store Conditionnal*

Avec ces deux instructions, on peut réaliser toutes les primitives de synchronisation.

- `ll` emplacement $->$ registre
- modifier la valeur dans le registre et éventuellement effectuer d'autres opérations
- `sc` registre $->$ emplacement

Le bus est « *verrouillé* » entre le `ll` et le `sc` et *aucun* basculement de thread ou de process ne peut avoir lieu.

[3] Appelé *load exclusive* dans l'architecture Arm, *load reserved* dans RISC-V, etc.

Mise en oeuvre de `ll/sc`

Le verrouillage physique du bus serait *très* pénalisant en termes de performances (interdiction de tout accès mémoire).
En fait, il suffit de vérifier les accès *à certaines adresses*.

Le processeur mémorise l'adresse du `ll` et espionne le bus.

Si un autre processeur modifie une donnée dans la ligne de cache concernée par le `ll` *avant* l'exécution du `sc`, le `sc` échouera et le processus devra être repris.

Cette méthode (`ll/sc` *faible*) est mise en oeuvre sur la plupart des processeurs RISC.

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Sur les processeurs ARM, `ldrex/strex` (ARMv6 et v7), et `ldxr/stxr` (ARM version 8).

`ldxr Wt, [Xn] ;` similaire à un `ld`, mais « marque » `Xn` comme accès exclusif

`stxr Ws, Wt, [Xn] ;` `Xn` adresse mémoire, `Wt`, registre à ranger, `Ws` reçoit 0 si succès (aucun accès à `Xn` entre le `ldxr` et le `stxr`), ou 1 si échec du store exclusif

Acquisition d'un verrou par TAS

```
1       ldr    r1, #1      ; r1 == 1 == locked
2  1    ldxr   r2, [r0]    ; load exclusive mutex @ address r0
3       cmp    r2, r1      ; Test if mutex is locked or unlocked
4       beq    %b1         ; If locked - retry (branch to '1')
5       stxrne r2, r1, [r0] ; Not locked, attempt to lock it
6       cmpne  r2, #1      ; Check if Store-Exclusive failed (1)
7       beq    %b1         ; stxr failed. Go to start of spinlock
8       ;; lock acquired
```

Support matériel des synchronisations et de l'exclusion mutuelle (cont.)

Les instructions réalisant des opérations atomiques ou les paires `ll/sc` impliquent une haie pour éviter des déplacements d'instructions.

- mauvaise utilisation du pipeline
- temps d'exécution de plusieurs dizaines à plusieurs centaines de cycles

Leur utilisation est impérative dans de nombreux programmes, mais elle peut facilement rendre un programme parallèle plus lent qu'un programme séquentiel...

Peuvent être programmées de manière portable.
En C, utilisation des fonctions de la `libc`: `<stdatomic.h>`
`atomic_thread_fence()`, `atomic_flag_test_and_set()`, `atomic_fetch_add()`, `atomic_exchange()`, `atomic_compare_exchange_weak()`, etc.
En C++, ils existe de même les fonctions/types `std::atomic` dans la STL.
Les variables partagées doivent être déclarées comme `volatile` pour éviter les optimisations du compilateur.

Table of content

1 Modéle de mémoire partagée

2 Instructions de synchronisation

3 Cohérence de cache
- Position du problème

4 Protocoles de cohérence
- Protocoles à mise à jour et protocoles à invalidation
- Protocole MSI
- Protocole MESI
- Protocoles à répertoire

Table of content

1 Modéle de mémoire partagée

2 Instructions de synchronisation

3 Cohérence de cache
- Position du problème

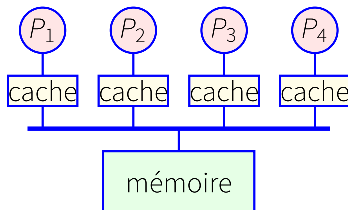
4 Protocoles de cohérence

Cohérence de cache : Position du problème

Les multicoeurs actuels sont dans une architecture SMP *symmetric multi-processor*

Ils ont tous accès à une mémoire commune *partagée*, ce qui permet de mettre en oeuvre un parallélisme de *thread*.

Par contre, l'accès à cette mémoire est lent, et chaque processeur est associé à un cache.



Cohérence de cache : Position du problème (cont.)

- Chaque cache est privé.
- Plusieurs copies d'une ligne de cache peuvent être présentes dans différents processeurs
- Une mise à jour locale
  - conduit à un état incohérent
  - que le cache soit à écriture simultanée (*write-through*) ou à réécriture (*write-back*)
- Dans un système à base de bus, l'information est globalement visible
- Dans un système à base de réseau sur puce (*Network-onChip NoC*), l'information n'est que locale

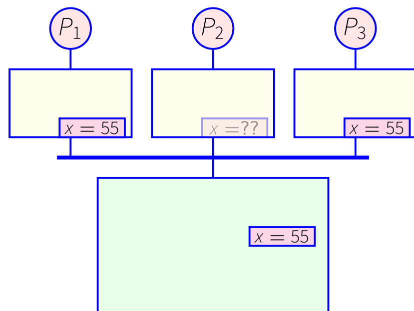
Cohérence de caches

Rappels : on distingue deux types de cache

Ecriture simultanée (*write-through*): toute écriture dans le cache est aussi effectuée en mémoire

Réécriture (*write back*) : on n'écrit que dans le cache, mais on marque la ligne comme *modifiée*. Quand la ligne est retirée du cache, on la réécrit en mémoire.

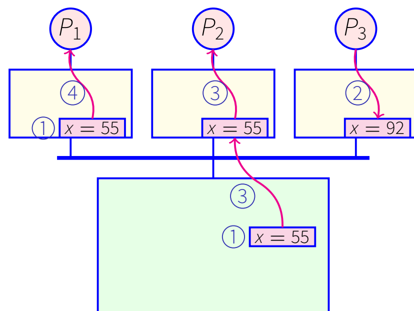
Exemple de scénario:



1. Initialement
   - `mem[x]` contient la valeur 55
   - Les caches de P_1 et P_3 contiennent également 55 à l'adresse x
   - Le cache de P_2 ne contient pas la ligne x

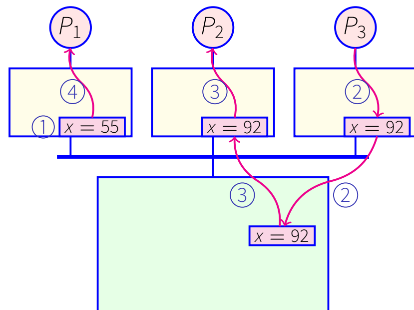
   Que se passe-t-il pour la séquence suivante :
2. P_3 écrit 92 à l'adresse x
3. P_2 lit x.
4. P_1 lit x.

Cas des caches *write-back*



1. Initialement
   - `mem`[x] contient la valeur 55
   - Les caches de P_1 et P_3 contiennent également 55 à l'adresse x
   - Le cache de P_2 ne contient pas la ligne x
2. P_3 écrit 92 à l'adresse x
   Comme le cache est *write-back*, rien n'est écrit en mémoire.
3. P_2 lit x. Comme le cache ne contient pas x, on lit la valeur incorrecte de la mémoire
4. P_1 lit x. x est dans le cache et c'est la valeur incorrecte du cache qui est envoyée à P_1.

Cas de caches *write-through*



1. Initialement
   - Les caches de P_1 et P_3 et la mémoire contiennent 55 à l'adresse x
   - Le cache de P_2 ne contient pas la ligne x
2. P_3 écrit 92 à l'adresse x
   Comme le cache est *write-through*, 92 est aussi écrit en **mem**[x].
3. P_2 lit x. Comme le cache ne contient pas x, lecture en mémoire de la valeur correcte 92
4. P_1 lit x. Comme x est présent dans le cache de P_1, c'est la valeur incorrecte du cache qui est envoyée eu processeur.

Table of content

1. Modéle de mémoire partagée
2. Instructions de synchronisation
3. Cohérence de cache
   - Position du problème
4. **Protocoles de cohérence**
   - Protocoles à mise à jour et protocoles à invalidation
   - Protocole MSI
   - Protocole MESI
   - Protocoles à répertoire

Table of content

1. Modéle de mémoire partagée
2. Instructions de synchronisation
3. Cohérence de cache
4. Protocoles de cohérence
   - Protocoles à mise à jour et protocoles à invalidation
   - Protocole MSI
   - Protocole MESI
   - Protocoles à répertoire

protocoles de cohérence

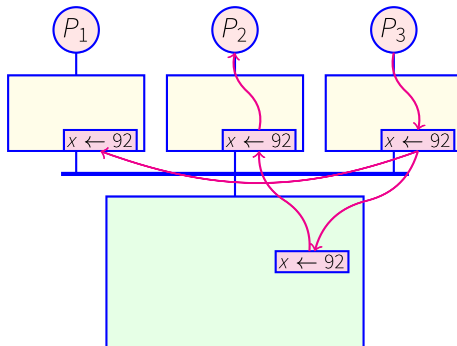
On suppose que l'on est dans un contexte de bus.

Tous les caches ont accès au bus et peuvent *espionner* les transactions (*bus snoop*)
Ils peuvent en déduire les données en cache correctes ou incorrectes en échangeant éventuellement des informations (*protocoles de cohérence de cache*)

Deux grandes classes de protocoles :

- protocoles à mise à jour
- protocoles à invalidation

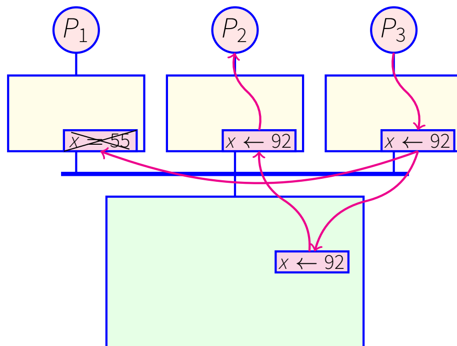
Protocole à mise à jour avec cache *write-through*



Chaque cache espionne le bus en permanence
Quand un cache voit une transaction en écriture concernant une donnée (ligne) qu'il possède, il recopie la donnée.

C'est ainsi que le cache de P_1 met à jour sa valeur quand il voit l'écriture mémoire de P_3

Protocoles à invalidation et cache *write-through*



Chaque cache espionne le bus en permanence
Quand un cache voit une transaction en écriture sur une donnée qu'il possède, il invalide sa donnée locale.
Quand il en aura besoin, il ira chercher la valeur en mémoire.

Ainsi la valeur dans P_1 est invalidée. A la prochaine lecture, P_1 lira la bonne valeur en mémoire

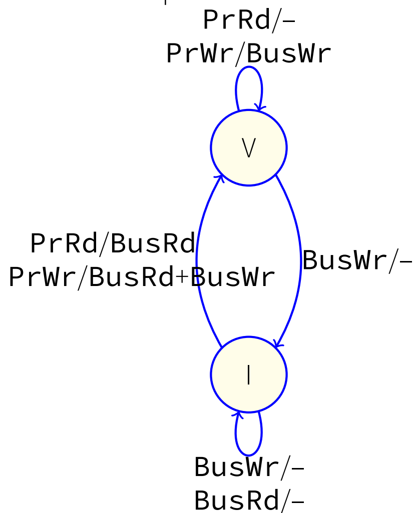
Protocoles à invalidation et cache *write-through* (cont.)

Ce mécanisme peut se décrire sous forme de protocole (automate).
Chaque ligne de cache peut avoir deux états : valide (V) ou invalide (I).
Il peut y avoir des actions du processeur ou du bus :

- Processeur :
  - `PrRd`: le processeur lit la donnée à l'adresse de la ligne
  - `PrWr`: le processeur écrit la donnée à l'adresse de la ligne
- Bus :
  - `BusRd`: il y une lecture de la donnée sur le bus
  - `BusWr`: il y a une écriture de la donnée sur le bus.

Protocoles à invalidation et cache *write-through* (cont.)

Protocole simple de cohérence pour un cache *write-through*.



Protocoles avec caches *write-back*

Les caches *write-through* génèrent un traffic très important sur les bus

Passage aux caches *write-back*

La plupart des écritures n'ont lieu qu'au niveau des processeurs

Comment espionner et assurer la cohérence ?

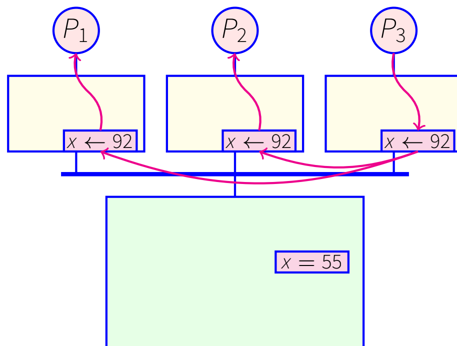
Les lignes de cache peuvent avoir plusieurs états indiquant si la ligne est valide, si elle est ou non dans plusieurs caches, si elle est cohérente avec la mémoire, etc.

Lors d'une transaction, l'état de la ligne de cache sera modifié dans les différents caches concernés en fonction d'un protocole.

Peut se faire avec mise à jour ou invalidation.

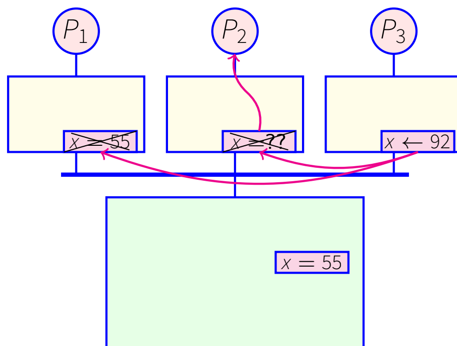
Protocole avec mise à jour et caches *write-back*

‘



Si la donnée est partagée, lors d’une modification le propriétaire devra la mettre à jour dans tous les autres caches utilisant la donnée
Peut génèrer un traffic très important en cas d’écritures multiples en mémoire.

Protocole avec invalidation et cache *write-back*



En cas d'écriture du processeur, le propriétaire de la donnée l'invalide dans les caches partageant cette donnée.
En cas de nouvelle écriture de cette donnée, une nouvelle invalidation sera inutile
Réduit sensiblement le traffic sur le bus

Si P_1 ou P_2 veulent accéder à x, le protocole devra assurer que la donnée lue sera celle du cache de P_3 et non celle de la mémoire.

Table of content

1. Modéle de mémoire partagée
2. Instructions de synchronisation
3. Cohérence de cache
4. Protocoles de cohérence
   - Protocoles à mise à jour et protocoles à invalidation
   - Protocole MSI
   - Protocole MESI
   - Protocoles à répertoire

Le protocole MSI

Pour mettre en oeuvre une invalidation avec cache *write-back*, utilisation d'un protocole avec plusieurs états pour chacune des lignes de cache.

Etats d'une ligne de cache :

I invalide

S partagée (*shared*): ce cache et éventuellement un ou plusieurs autres ont une copie valide et elle est identique à celle de la mémoire

M modifiée : seul ce cache a une copie valide (et la donnée en mémoire est invalide)

L'automate associé à chaque ligne va changer d'état et le cache effectuer un certain nombre d'opérations, en fonction de l'état initial de la ligne de cache, des actions du processeur ou des événements sur le bus.

Le protocole MSI (cont.)

Les événements processeur :

- `PrRd`(lecture à l'adresse de la ligne)
- `PrWr`(écriture à l'adresse de la ligne)

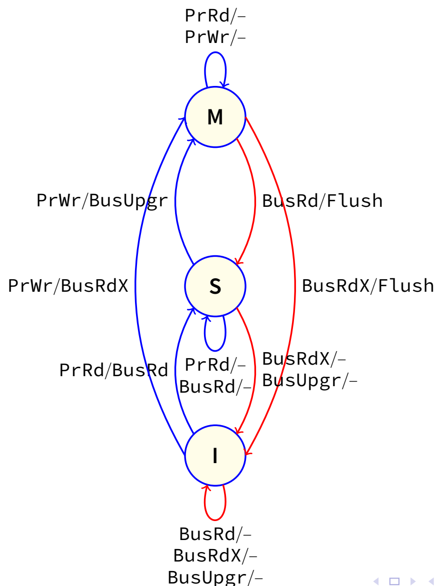
Les transactions bus :

- `BusRd`: demande une copie sans vouloir la modifier (par exemple suite à un échec en lecture)
- `BusRdX`: demande une copie pour la modifier (par exemple suite à un échec en écriture)
- `BusUpgr`: invalide la ligne dans les autres caches (par exemple suite à un succès en écriture)
- `Flush`: copie la ligne dans la mémoire

Actions du cache :

- change d'état
- émet une transaction bus (`BusRd`, `BusRdX`, `BusUpgr`, `Flush`)

L'automate MSI



Si état = *shared*

- une écriture locale rend l'état *modified*
- une écriture distante rend l'état *invalid*

Si état = *Modified*

- en cas de lecture distante en échec, on fait un *write back* (`Flush`), et l'état devient *shared*
- en cas d'écriture distante, l'état devient *invalid*

Si état = *Invalid*

- une lecture locale en échec rend l'état *shared*
- une écriture locale en échec rend l'état *Modified*

A partir d'un état (I)

Actions du bus : sans effet

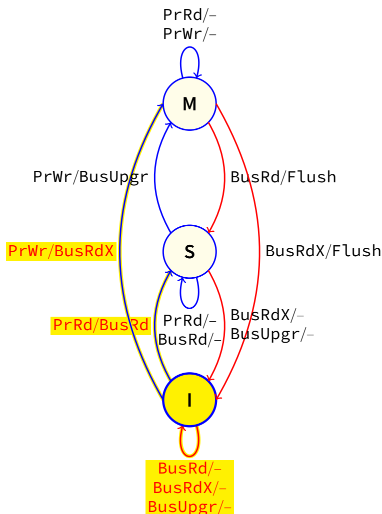
Actions du processeur :

PrWr passage à un état (M)

— émission d'un BusRdX
la mémoire envoie la donnée, mais si un autre cache est dans un état (M), la lecture mémoire est annulée et il envoie le contenu de la ligne

PrRd passage à un état (S)

— émission d'un BusRd
la mémoire envoie la donnée, mais si un autre cache est dans un état (M), la lecture mémoire est annulée et ce cache envoie le contenu de la ligne



Le protocole MSI (cont.)

A partir d'un état (S)

Actions du processeur :

PrRd sans effet

PrWr passage à un état (M)

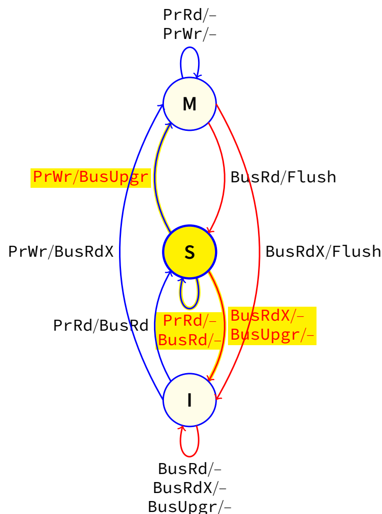
— émission d'un BusUpgr pour invalider les (éventuels) autres caches à l'état (S)

Actions du bus :

BusRd pas de changement

BusRdX ou BusUpgr passage à un état (I)

— comme la mémoire est cohérente avec le(s) cache(s), aucune autre action n'est nécessaire



A partir d'un état (M)

Actions du processeur : aucun changement

Actions du bus :

BusRd passage à un état (S)

- On interrompt la lecture mémoire
- le contenu de la ligne est envoyé vers la mémoire par un `Flush` (et lu par le cache ayant émit un `BusRd`)

BusRdX passage à un état (I)

- On interrompt la lecture mémoire
- le contenu de la ligne est envoyé vers la mémoire par un `Flush` (et lu par le cache ayant émit le `BusRdX`)

(un `BusUpgr` est impossible car les autres caches sont dans un état (I))

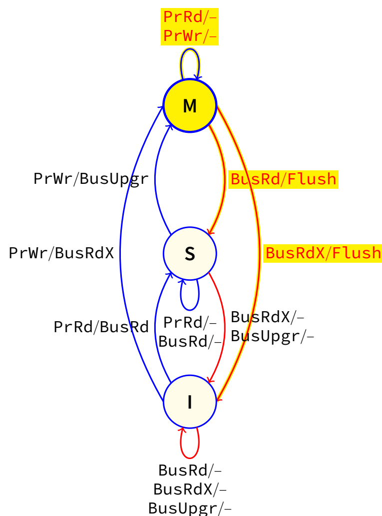


Table of content

1 Modéle de mémoire partagée

2 Instructions de synchronisation

3 Cohérence de cache

4 Protocoles de cohérence
- Protocoles à mise à jour et protocoles à invalidation
- Protocole MSI
- Protocole MESI
- Protocoles à répertoire

Le protocole MESI

Dans MSI, un cache avec une ligne dans l'état S ne peut pas savoir si d'autres caches ont une copie.
Ceci entraîne en cas d'écriture (passage à un état M) d'inutiles transactions sur le bus pour invalider les autres caches (`BusUpgr`)
Par exemple, dans le cas (très courant) où un processeur lit une donnée (et il est le seul) puis la modifie.

`ld @X` le cache fait un `BusRd` pour acquérir la donnée depuis la mémoire, et passe à l'état S

`st @X` pour modifier la donnée, il fait un `BusUpgr` pour (éventuellement) invalider la donnée dans les autres caches et il passe à l'état M.

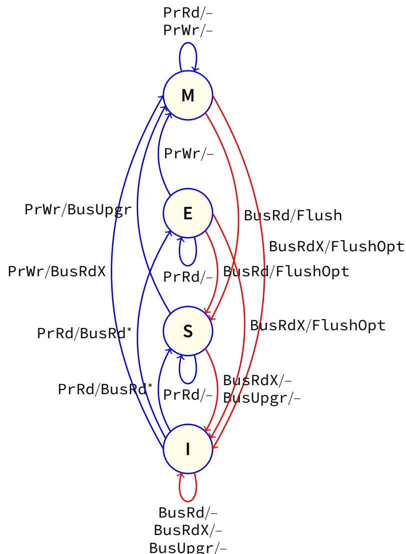
Mais, comme le cache est le seul à posséder la donnée, c'est une action inutile.

Introduction d'un état E (*exclusif*)
Correspond à l'état S, quand un seul cache possède une copie cohérente avec la mémoire

Le protocole MESI (cont.)

- 4 états
  - **E** Exclusif
    - Seule copie de la ligne
    - Non modifié (= Mémoire)
  - **M** Modifié (et != mémoire)
  - **S** Partagé (*shared*)
    - la ligne existe à l'identique dans au moins un autre cache
    - identique en mémoire
  - **I** Invalide
- Actions
  - Processeurs :
    - **`PrRd`**
    - **`PrWr`**
  - Bus :
    - **`BusRd`** demande de lecture
    - **`BusRdX`** demande de lecture pour modification
    - **`BusUpgr`** invalidation du cache par un autr processeur
    - **`Flush`** copie d'une ligne en mémoire
    - **`FlushOpt`** copie d'une ligne *vers un autre cache*

Le protocole MESI (cont.)



A partir d'un état (I)

Actions du bus : sans effet

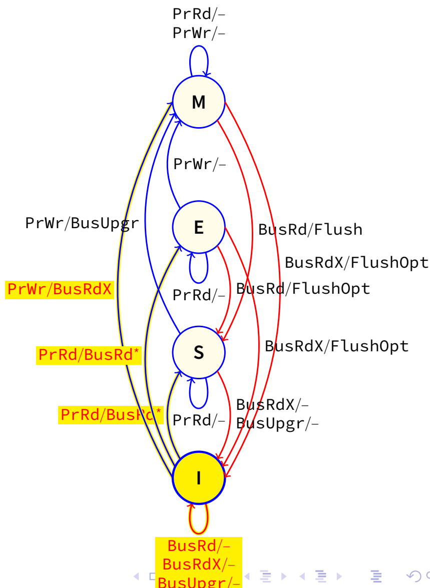
Actions du processeur :

PrWr passage à un état (M)

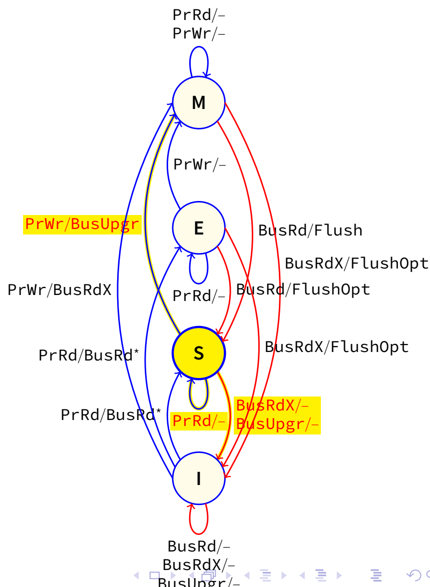
- émission d'un BusRdX
  la mémoire envoie la donnée, mais
  si un autre cache est dans un état (M), la lecture mémoire est annulée et il envoie le contenu de la ligne

PrRd émission d'un BusRd

- si un cache répond, passage à un état (S) (plusieurs copies)
- si la mémoire répond, passage à un état (E) (unique copie)



A partir d'un état (S)



Actions du processeur :

PrRd pas d'effet

PrWr passage à un état (M)
émission d'un BusUpgr pour invalider les autres caches

Actions du bus :

BusRdX ou BusUpgr on passe à un état (I)

Le protocole MESI (cont.)

A partir d'un état (E)

Actions du processeur :

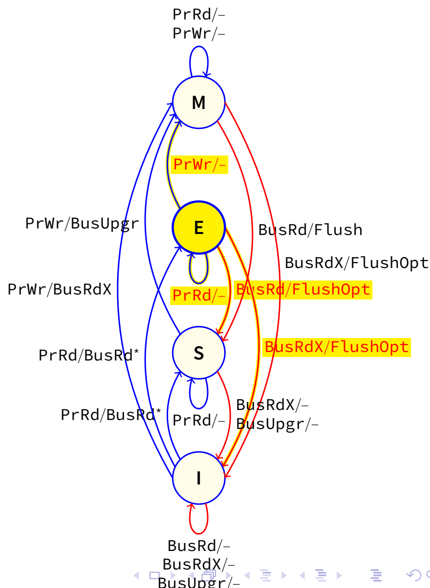
PrRd pas d'effet

PrWr passage à un état (M)

Actions du bus :

BusRdX on passe à un état (I)
on envoie la ligne au cache concerné (`FlushOpt`)

BusRd on passe à un état (S)
on envoie la donnée cache concerné (`FlushOpt`)
(par definition, dans un état (E), la ligne est déja cohérente avec la mémoire)



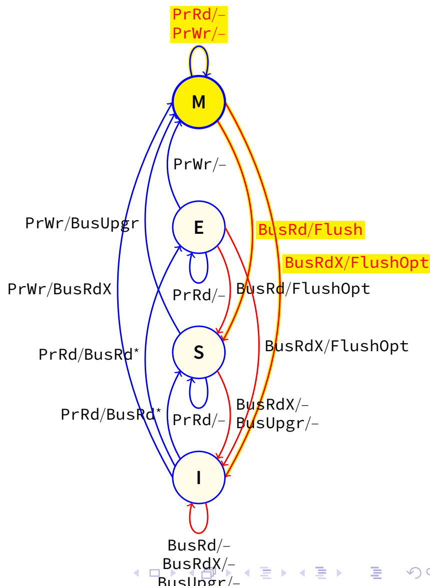
A partir d'un état (M)

Actions du processeur : pas d'effet

Actions du bus :

BusRd on passe à un état (S)
on envoie la ligne au cache concerné et on met à jour la mémoire (`Flush`)

BusRdX on passe à un état (I)
on envoie la ligne au cache concerné (`FlushOpt`) (la ligne va être modifiée et la cohérence mémoire–cache n'est donc pas requise)



Protocole MOESI

Améliorations de l'algorithme pour limiter les accès à la mémoire.

MOESI : ajout d'un état O (*owner*) qui indique un état à la fois modifié et partagé.

Owner indique que la copie locale est sale (modifiée) et donc que la mémoire a une valeur incorrecte.
Mais à la différence de M, il existe des copies dans d'autres caches (qui seront dans l'état *shared*).

Les caches dans l'état *shared* n'ont pas le droit de modifier leur valeur (contrairement à MESI).
Ils doivent préalablement devenir *owner* (*request for ownership*)

Seul le cache qui est *owner* peut mettre à jour la donnée.

Il enverra la valeur aux autres caches si nécessaire.

Permet de réduire le nombre d'accès à la mémoire.

C'est le protocole utilisé dans les processeurs intel.

Protocole MOESI (cont.)

Une autre variation est le protocole MESIF.

Introduction d'un état F (*forward*).

Contrairement à l'état *owner* de MOESI, les lignes dans l'état *forward* sont cohérentes avec la mémoire (propres).

L'état F permet d'éviter que plusieurs caches répondent quand une donnée mémoire est demandée.
Seul le cache F donnera la donnée au cache.

MESIF est le protocole utilisé dans les processeurs AMD.

MOESI permet de partager rapidement des lignes de cache modifiées, alors que MESIF permet de partager rapidement des lignes *non* modifiées.
Il existe aussi un protocole MOESIF.

Cas des caches multiniveaux

Comment garantir la cohérence dans le cas d'une hiérarchie de caches (L1, L2, etc) ?

- Tous les caches participent au protocole...
- utilisation de caches *inclusifs*.

Dans ce cas, toutes les lignes de L1 sont aussi dans L2, et L2 peut entièrement gérer le protocole.

On ajoute dans l'état L2 une information indiquant si la ligne est dans L1.

Il faut traiter le cas d'un succès en écriture dans L1 si L1 n'est pas à écriture simultanée (*write through*). Ajout d'un état indiquant que la donnée est modifiée, mais uniquement présente dans L1.

Il faut propager le passage à l'état *invalid* de L2 à L1.

Faux partage

Si on ne prend pas de précaution, les protocoles de cohérence peuvent générer un traffic inutile important.
Cas du *faux partage* (*false sharing*)

Exemple : calcul de plusieurs réductions

```
1 double vecteurs[k][N];
2 double sommes[k];
3 #pragma omp parallel for
4 for(int i=0; i<k; i++)
5   for(int j=0; i<N; j++)
6     sommes[i] += vecteurs[i][j] ;
```

Les différents `sommes[i]` sont modifiés par des threads différents, donc pas de nécessité d'accès atomique
mais ils appartiennent possiblement tous à la *même* ligne de cache
Chaque écriture va provoquer un `Flush` dans le cache modifié en dernier et une invalidation.

Traffic important et inutile

La bonne solution est d'utiliser une variable locale à chaque thread.
Mise à jour du vecteur `sommes[]` unique à la fin.

```
double vecteurs[k][N];
double sommes[k];
#pragma omp parallel for
for(int i=0; i<k; i++) {
  double tmp=0.0 ; // une variable locale (privée)
  for(int j=0; i<N; j++)
    tmp += vecteurs[i][j] ;
  // une fois la somme faite, recopie dans
  // le vecteur sommes
  sommes[i] = tmp;
}
```

Le faux partage est une cause importante d'inefficacité des programmes parallèles.

Table of content

1 Modéle de mémoire partagée

2 Instructions de synchronisation

3 Cohérence de cache

4 Protocoles de cohérence
- Protocoles à mise à jour et protocoles à invalidation
- Protocole MSI
- Protocole MESI
- Protocoles à répertoire

Cas des systèmes manycores

Assurer une mémoire entièrement partagée ne peut s'envisager quand le nombre de processeurs augmente.

On passe alors à une architecture DSM (*distributed shared memory*) également appelée cc-NUMA (cache coherent - Non Uniform Memory Access).

Chaque processeur a un cache et une mémoire locale et peut communiquer avec les autres processeurs par un *Network-on Chip NoC*.

Les processeurs peuvent accéder à toutes les mémoires, mais l'accès à la mémoire locale est beaucoup plus rapide.

On peut utiliser les mécanismes de *snoop* (envoi de toutes les transactions à tous les processeurs), mais cela implique d'émuler le comportement d'un bus sur le NoC.

On utilise fréquemment un protocole à base de répertoire (*directory*).

Un répertoire (*directory*) gére chacune des lignes de cache. Il peut être centralisé ou situé dans le banc mémoire où est stockée la ligne.
Dans le répertoire, chaque ligne a un état :

- **uncached** (U) la donnée est uniquement en mémoire
- **shared** (S) la donnée est dans un ou plusieurs processeurs et en mémoire
- **exclusive/modified** (EM) la donnée est valide dans un processeur et la valeur en mémoire est incorrecte

Dans les cas S ou EM, le répertoire indique le propriétaire.
Dans le cas S, le répertoire indique également toutes les lignes possédant une copie valide.
Exemple avec 4 processeurs P0, P1, P2 et P3

ligne	état	*owner*	P0	P1	P2	P3
0	U					
1	S	P2	0	1	1	0
2	EM	P1	0	1	0	0
3	S	P0	1	1	1	0

Cas des systèmes manycores (cont.)

Dans le cas d'une requête, le processeur concerné (*requestor*) va :

- demander au répertoire l'état de la ligne et son propriétaire s'il ne les connait pas déjà
- gérer un protocole de cohérence de type MOESI entre *requestor* et *owner* (et éventuellement les processeurs possédant aussi une copie de la ligne (*sharers*)).
- informer le répertoire des changements éventuels d'état ou de propriétaire

Toutes les communications se passent par passage de message sur le NoC
Permet de limiter le traffic.

La difficulté provient des courses éventuelles entre requêtes sur le NoC et des problèmes de consistence de la mémoire.

Cas des systèmes manycores (cont.)

Les systèmes manycores abandonnent parfois la cohérence de cache systématique.

La cohérence, si nécessaire, est alors gérée par le compilateur (ou le programmeur) avec des instructions spécifiques de cohérence.

Problème difficile à cause des possibilités d'alias mémoire.

Peut être simplifié si le compilateur connaît l'ensemble des opérations effectuées par tous les processeurs (open-MP).

L'intérêt est de mieux maitriser les temps d'accès mémoire et de les rendre plus déterministes pour des systèmes temps-réel.