Machines parallèles à mémoire distribuée: protocoles de cohérence de caches et modèles de consistence mémoire

Guy Tremblay
Dépt. d'informatique
UQAM

Juillet 1996

0. Aperçu

1. Optimisations intéressantes dans les machines uniprocesseurs modernes
2. Évolution des architectures parallèles
3. Protocoles de cohérence de cache
4. Modèles de *consistence* de mémoire
5. Conclusion/avenues de recherche

1. Optimisations intéressantes dans les machines modernes

Réordonnancement des instructions par le compilateur

Motivation: Le plus tôt les chargements sont initiés, le plus tôt les opérandes seront disponibles.

Optimisation correcte sur une machine uni-processeur lorsque deux accès mémoires vont à des adresses différentes

```
store $2, 2300($6)
store $3, 2304($6)
load  $1, 1000($6)
add   $1, $3, $5
```

⇒

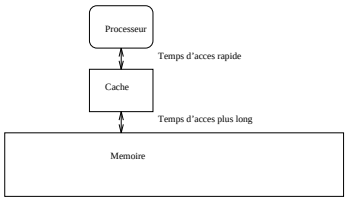
```
load  $1, 1000($6)
store $2, 2300($6)
store $3, 2304($6)
add   $1, $3, $5
```


Utilisation de caches

Objectif: diminuer le temps de latence des accès mémoire

Cache:

Contient copie des informations récemment accédées



Adresse désirée déjà dans cache

⇒ temps accès ≈ temps cycle de machine

Adresses désirée pas dans cache (faute de cache)

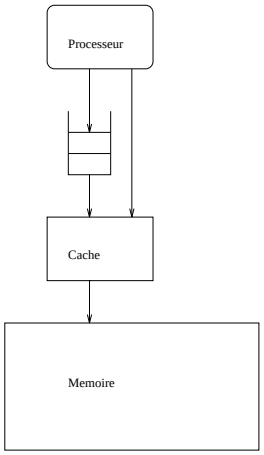
⇒ temps accès ≈ 5-10 cycles machine

⇒ **gel** du pipeline du processeur (*stall*)

Tampon d'écriture et exécution pipelinée des accès mémoire

Exécution d'un `store`: Le processeur *amorce* l'exécution du `store` puis continue immédiatement avec l'instruction suivante

⇒ Plusieurs accès mémoire en cours d'exécution

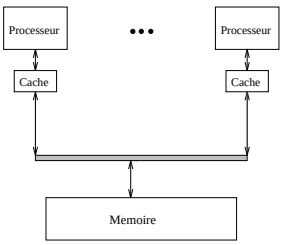



```
store $2, 100($6)        ; Complète immédiatement
store $4, 104($7)        ; Complète immédiatement
load $3, 104($7)         ; Attente pour la valeur
```


2. Évolution des architectures parallèles (MIMD)

- Machine MIMD avec bus et mémoire partagée
- Machine MIMD avec réseau et mémoire distribuée
- Machine multi-contextes

2.1 Machine MIMD avec bus et mémoire partagée

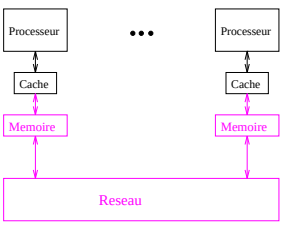


Avantages/désavantages:

\+ Problème de la cohérence des caches facilement résolu (*snoopy bus*)

– Ne permet pas le parallélisme massif (≈ 30 processeurs)

2.2 Machine MIMD avec réseau et mémoire distribuée



Avantages/désavantages:

\+ Permet un plus grand parallélisme (transactions multiples sur le réseau)

= Problème de la cohérence des cache résoluble mais plus complexe

– Temps de latence imprévisible ⇒ Que faire en cas de faute de cache/accès réseau?

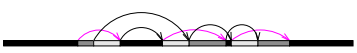
2.3 Machine multi-contextes (*multithreaded computer*)

Processeur multi-contextes =

- Contexte (*thread*) = `FP + IP + registres`
- Conserve plusieurs contextes indépendants ⇒ (presque) toujours plusieurs instructions prêtes à s'exécuter
- Changement de contexte rapide et peu coûteux ⇒ Changement de contexte en cas d'accès réseau

Sans changement de contexte:

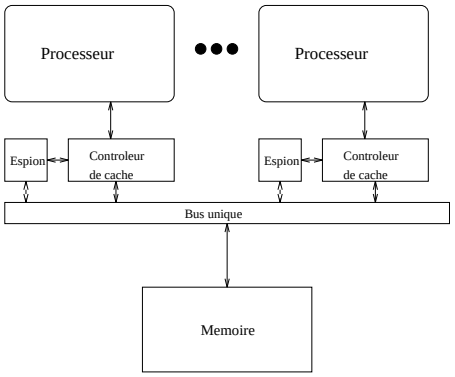
Avec changement de contexte:



3. Protocoles de cohérence de cache

3.1 Protocole pour bus: Espionnage

Espionnage = *snoopy bus*



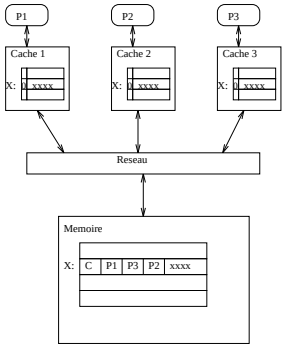
- Espion écoute le bus pour déterminer s'il a ou non une copie d'un bloc transmis sur le bus:
  - Possède *une* copie en lecture d'un bloc demandé pour écriture: le bloc est invalidé dans cache
  - Possède *la* copie en écriture demandé pour lecture: le bloc est écrit en mémoire (sera lu par demandeur) puis invalidé dans cache

3.2 Protocole pour réseau: *directory*

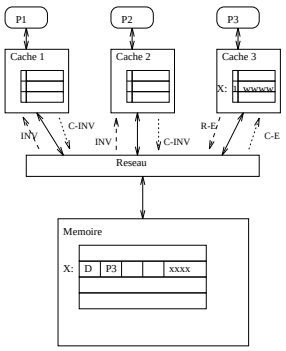
Une machine utilisant un réseau ne peut pas utiliser espionnage pcq. requiert *broadcasting* (des invalidations) ⇒ trop coûteux sur réseau

Solution alternative: utilisation de répertoires (*directory scheme*)

- Un répertoire est associé à chaque module de mémoire
- Le répertoire contient une liste de *pointeurs vers les caches* qui ont une copie d'un bloc provenant du module mémoire associé



Suites d'événements dans le cas où le processeur `P3` désire écrire à l'adresse `X`:



(1) R-E = Requete Ecriture (acces Exclusif) (=> P3 bloque (stalled))

(2) INV = INValidation

(3) C-INV = Confirmation INValidation

(4) C-E = Confirmation Ecrirure

1. Le cache 3 détecte que le bloc est valide mais que l'accès en écriture exclusive n'est pas permis (lecteurs multiples).
2. Le cache 2 envoie une requête d'écriture à la mémoire et suspend (*stall*) le processeur 3.
3. Le module mémoire envoie des requêtes d'invalidation aux caches 1 et 2.
4. Les caches 1 et 2 recoivent les requêtes d'invalidation, invalident les blocs et retournent une confirmation.
5. Une fois toutes les confirmations reçues par le module mémoire, celui-ci envoie la permission d'écriture à la cache 3.
6. La cache 3 reçoit la permission d'écriture, met à jour le statut du bloc mémoire et réactive le processeur P3.

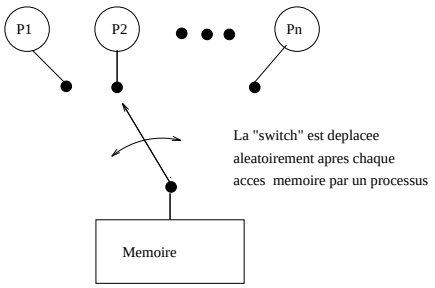
4. Modèles de consistence mémoire

4.1 Modèle de consistence séquentielle

Modèle de consistence mémoire séquentielle

= Nécessite que l'exécution d'un programme parallèle apparaisse comme un *entrelacement* (*interleaving*) de l'exécution des divers processus parallèles sur une machine séquentielle.

⇒ tous les accès mémoire semblent s'exécuter dans le même ordre pour tous les processeurs



Exemple

Le programme suivant est-il correct?

P1
```
mailbox = message1;
mailbox_full = TRUE;
```

P2
```
while (!mailbox_full)
   ;
m = mailbox;
/* ?  m == message1 ?  */
```

Réponses possibles:

- Oui: si un modèle de consistence séquentielle de mémoire est utilisée
- Non: ...

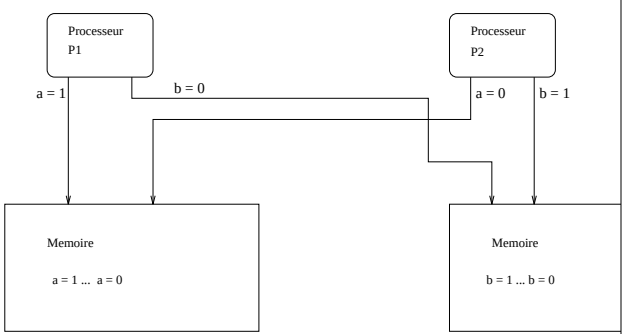
Exemple

	P1		P2
I_1 :	`b = 0`	I_3 :	`a = 0`
I_2 :	`a = 1`	I_4 :	`b = 1`

Résultats possibles selon le modèle de consistence mémoire séquentielle:

$I_1 \to I_2 \to I_3 \to I_4 \quad \Rightarrow \quad a = 0 \quad b = 1$
$I_1 \to I_3 \to I_2 \to I_4 \quad \Rightarrow \quad a = 1 \quad b = 1$
$I_1 \to I_3 \to I_4 \to I_2 \quad \Rightarrow \quad a = 1 \quad b = 1$
$I_3 \to I_1 \to I_2 \to I_4 \quad \Rightarrow \quad a = 1 \quad b = 1$
$I_3 \to I_1 \to I_4 \to I_2 \quad \Rightarrow \quad a = 1 \quad b = 1$
$I_3 \to I_4 \to I_1 \to I_2 \quad \Rightarrow \quad a = 1 \quad b = 0$

Mais:



Mise en oeuvre et coûts de consistence séquentielle

- Tous les accès mémoires d'un processeur doivent se compléter dans l'ordre dans lequel ils apparaissent dans le programme (et cet ordre doit être le même pour tous les processeurs)
- Nécessite de retarder l'émission d'un accès mémoire jusqu'à ce que les accès précédents aient été complétés

  ⇒ Écriture doit être confirmée avant que lecture ou écriture subséquente puisse s'amorcer donc mémoire doit retourner confirmation

Implications:

- Augmente le temps d'exécution des programmes
- Augmente le traffic sur le réseau

4.2 Modèles plus faibles de consistence mémoire

Objectif = Améliorer les performances des machines parallèles en permettant d'utiliser la plupart des optimisations mentionnées précédemment

Stratégie = Assurer consistence séquentielle uniquement pour *certaines* instructions spéciales de synchronisation et non pour toutes les instructions:

⇒ Coûts de consistence séquentielle associées uniquement à ces instructions spéciales

⇒ Instructions ordinaires d'accès peuvent s'exécuter rapidement (sans tenir compte de consistence)

Demande une stratégie différente d'écriture des programmes qui tient compte de synchronisation:

1. Identifier les endroits où des résultats inconsistents peuvent être produits (accès multiples à des locations mémoire partagées)
2. Insérer suffisamment d'instructions de synchronisation pour prévenir les effets indésirés

Exemple = *weak consistency*

P1
```
test&set(lock);
mailbox = message1;
mailbox_full = TRUE;
unset(lock);
```

P2
```
while (!mailbox_full)
   ;
test&set(lock);
m = mailbox;
unset(lock);

/* m == message1 */
```

Traitement des instructions de synchronisation:

1. Une instruction de synchronisation ne peut s'amorcer que lorsque les instructions de synchronisation précédentes ont globalement complété.
2. Une instruction de synchronisation ne peut s'amorcer que lorsque *tous* les accès mémoire précédents sont *globalement* complétés.
3. Les accès qui suivent ne peuvent s'amorcer que lorsque l'instruction de synchronisation a elle-même *globalement* complété

Écritures ⇒ confirmation que l'écriture a complété doit avoir été reçue (invalidation des caches!)

Exemple: *release consistency*

Encore moins contraignant que *weak consistency*:

P1
```
acquire(lock);
mailbox = message1;
mailbox_full = TRUE;
release(lock);
```

P2
```
while (!mailbox_full)
   ;
acquire(lock);
m = mailbox;
release(lock);

/* m == message1 */
```

Traitement des instructions de synchronisation

- `acquire`: Retarde l'exécution des accès mémoire ordinaires jusqu'à ce que le *acquire* soit globalement complété
- `release`: Ne s'exécute que lorsque tous les accès précédents ont globalement complété

Weak (WC) **vs.** *Release consistency (RC)*

Synchronisation dans *WC*:

`test&set` et `unset` ...

(A) Ne s'exécutent que lorsque tous les accès précédents ont complété

(B) Empêchent les accès qui suivent de s'amorcer tant que la synchronisation n'est pas complétée

Synchronisation dans *RC*:

(A) `release` ne s'exécute que lorsque tous les accès précédents ont complété

(B) `acquire` empêche les accès qui suivent de s'amorcer tant que la synchronisation n'est pas complétée

Corollaire pour *RC*:

- Les opérations après le `release` peuvent se compléter *avant* ceux qui précèdent le `release`
- Les opérations avant le `acquire` peuvent se compléter *après* le `acquire`

Mise en oeuvre de modèle RC:

- Suspendre le pipeline lors de l'exécution d'un `acquire` jusqu'à ce que le `acquire` complète
- Retarder la finalisation d'un `release` jusqu'à ce que tous les accès qui précèdent complètent:

  ⇒ Demande de recevoir une confirmation des écritures

  ⇒ Ajout de nouveaux éléments dans le tampon d'écriture doit être suspendu jusqu'à ce qu'il soit vidé

  Coûts matériel associés:

  - cache *lock-up free*
  - gestion de l'information sur les requêtes multiples en cours de traitement (*multiple outstanding requests*)

19-1

5. Conclusion/avenues de recheche

Encore peu de travaux sur la vérification des protocoles de cohérence de cache:

- Machines à états finis
- Réseaux de Petri
- Machines à états finis avec états symboliques utilisant la propriété que le nombre exact de copies dans les caches n'a pas d'importance: 0, 1 vs. plusieurs (pour éviter le problème de l'explosion du nombre d'états)

Nouveau modèle de consistence mémoire proposé par Gao & Sarkar (*location consistency*) basé sur une spécification d'un ordre *partiel* entre les opérations de lecture et/ou écriture (*pomset = partially ordered multiset*)

Protocole proposé par Shamir: Aucune vérification formelle effectuée

- Permet, contrairement aux autres approches et protocoles, d'avoir plusieurs *propriétaires* d'une location mémoire (*multiple-owner*), et ce même en écriture

20