Cohérence de la hiérarchie de mémoire partagée

Module 4

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

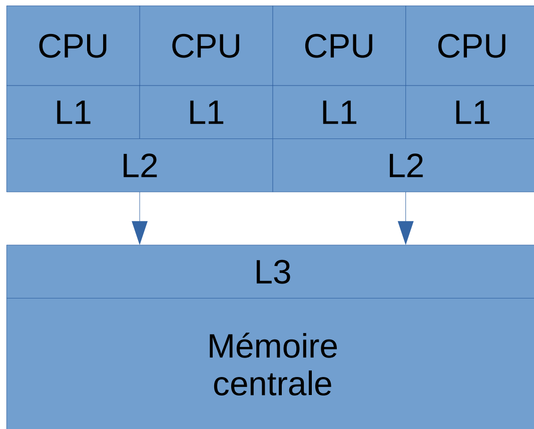
1. Introduction
2. Cohérence des caches
3. Architecture des caches
4. Interface logicielle
5. Optimisations
6. Modèles d'ordonnancement des accès mémoire

Cohérence de la hiérarchie de mémoire partagée

1. Introduction
2. Cohérence des caches
3. Architecture des caches
4. Interface logicielle
5. Optimisations
6. Modèles d'ordonnancement des accès mémoire

Cache et multi-processeurs

- Comment sont vues les écritures en mémoire d'un processeur à l'autre?
- Beaucoup d'autres organisations possibles!



Cohérence de la hiérarchie de mémoire partagée

1. Introduction
2. Cohérence des caches
3. Architecture des caches
4. Interface logicielle
5. Optimisations
6. Modèles d'ordonnancement des accès mémoire

Cohérence des caches

- Le Processeur P1 écrit n dans X puis lit X et trouve n.
- Le processeur P2 écrit n dans X puis le processeur P1 lit X et trouve n si assez de temps est écoulé.
- Le processeur P1 écrit n1 dans X et le processeur P2 écrit n2 dans X en même temps. La même valeur finale (n1 ou n2) sera vue par tous les processeurs.
- Les écritures sont propagées de manière asynchrone pour des fins de performance.

Propagation des écritures

- Réécriture: la valeur est écrite en cache puis copiée ultérieurement en mémoire centrale. Avertir que le bloc a été modifié pour empêcher toute autre modification simultanée!
- Ecriture immédiate: la valeur est immédiatement écrite en mémoire centrale, visible par tous.

Mise à jour de la cache

- Cohérence par mise à jour: une copie de la nouvelle valeur est envoyée à chaque copie en cache.
- Cohérence par invalidation: avertir, lorsqu'un bloc est modifié, de détruire les copie maintenant invalides.
- Statut d'un bloc en cache: Invalide, partagé (pour lecture seulement), exclusivité (peut être écrit et lu).

MESI, MOESI

- Modified: cette cache a la copie la plus récente, la seule, la copie en mémoire est invalide.
- Owned: cette cache a la copie la plus récente, d'autres peuvent avoir le statut Shared, la copie en mémoire peut être invalide.
- Exclusive: cette cache a la copie la plus récente, la seule, la copie en mémoire est valide.
- Shared: cette cache a la copie la plus récente, d'autres aussi, la copie en mémoire peut être invalide si une autre cache a le statut Owned.
- Invalid: ce bloc ne peut être utilisé, son contenu n'est plus à jour.

Cohérence par surveillance du bus

- Très populaire pour peu de processeurs, souvent avec écriture simultanée.
- Le message de mise à jour ou invalidation est envoyé sur le bus, les autres caches incorporent cette information si elle contiennent le bloc visé dans leur cache.
- L'accès au bus sérialise les accès.

Cohérence par répertoire

- Pour chaque bloc, liste des processeurs qui en ont une copie (e.g. vecteur de 8, 16, 32 bits...) et statut (modifié).
- Lorsqu'un bloc est modifié, s'il était en mode partagé, le répertoire est averti, un message d'invalitation est envoyé aux autres copies et le statut devient modifié.
- Pour accéder un bloc modifié, il faut prendre la nouvelle copie et il devient partagé, non modifié.
- Le répertoire peut être réparti avec une fonction de correspondance directe.

Surveillance versus répertoire

- Surveillance du bus: plus simple, était utilisé pour deux ou 4 processeurs, les processeurs actuels saturent trop rapidement le bus.
- Répertoire: plus compliqué mais requiert moins de communication.

Cohérence de la hiérarchie de mémoire partagée

1 Introduction

2 Cohérence des caches

3 Architecture des caches

4 Interface logicielle

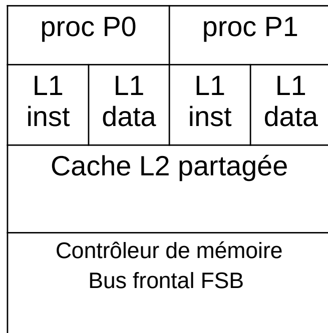
5 Optimisations

6 Modèles d'ordonnancement des accès mémoire

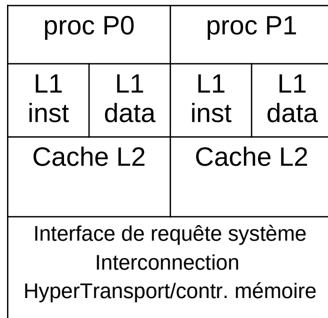
Architecture des caches

- Achitectures NUMA avec répertoire réparti; chaque processeur s'occupe d'une partie de mémoire centrale et du répertoire associé.
- Processeurs multi-coeurs avec caches partagées de manière différente d'un niveau à l'autre.

Exemple: Intel Core Duo et AMD Opteron

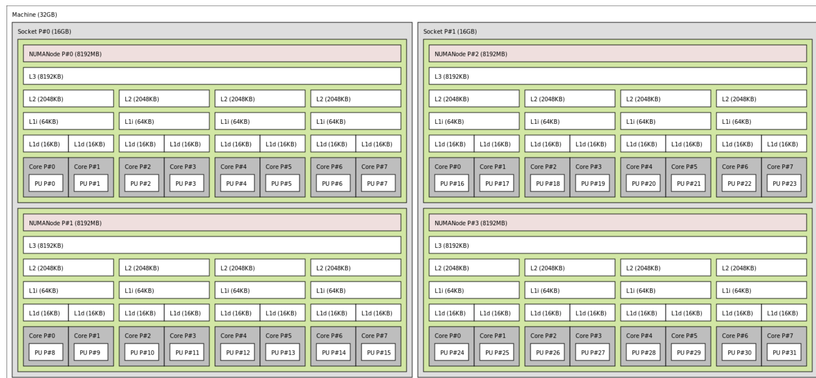


Intel Core Duo
Protocole MESI

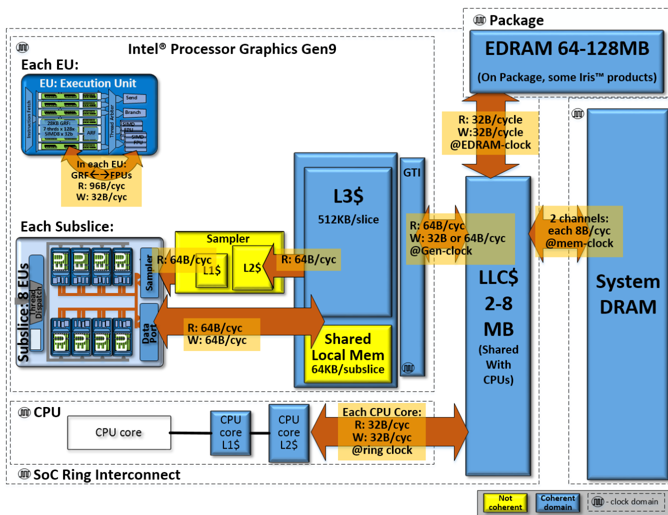


AMD Dual Core Opteron
Protocole MOESI

Exemple: AMD Bulldozer



Exemple: Intel Skylake



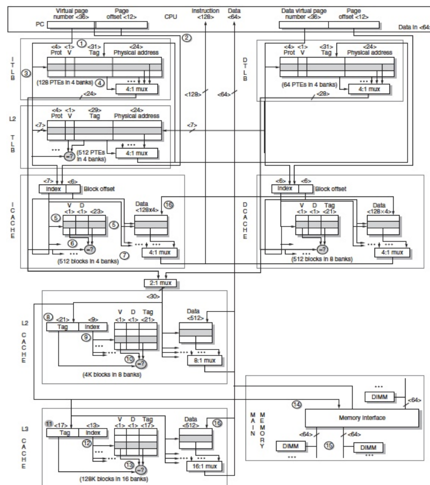
Exemple: Intel i7

	TLB instructions	TLB données	TLB niveau 2
Taille	128 entrées	64 entrées	512 entrées
Associativité	4	4	4
Remplacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Latence d'accès	1 cycle	1 cycle	6 cycles
Latence de faute	7 cycles	7 cycles	> 100 cycles

Exemple: Intel i7

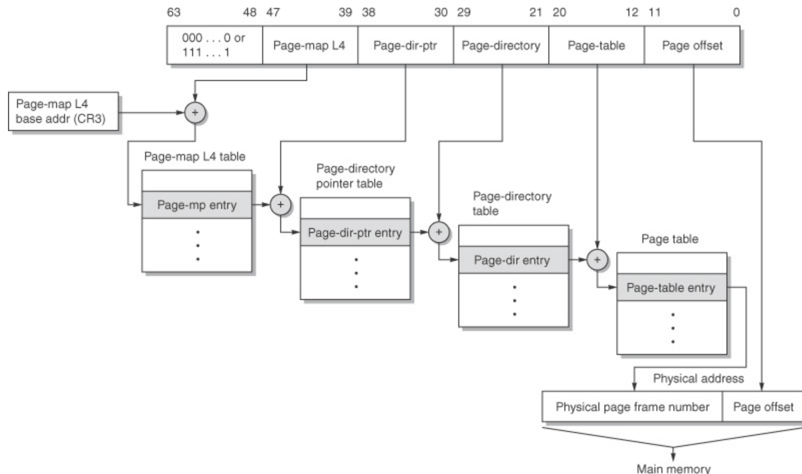
	Cache L1	Cache L2	Cache L3
Taille	32Kio I / 32Kio D	256Kio	8Mio (4 coeurs)
Associativité	4 I / 8 D	8	16
Remplacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Latence d'accès	4 cycles	10 cycles	35 cycles
Index	Index virtuel Etiquette physique	Index/étiquette physiques	Index/étiquettes physiques
Adresses	Adresses physiques 36 bits et adresses virtuelles 48 bits (# de page 36 bits et index dans la page 12 bits)		
Cohérence	Cache L3 inclusive avec répertoire de tous les blocs, leur statut et la liste des coeurs qui en ont une copie		

Exemple: Intel i7



Tiré du livre "Computer Architecture" de Hennessy et Patterson, 5ème édition.

Exemple: Intel i7



© 2007 Elsevier, Inc. All rights reserved.

Tiré du livre “Computer Architecture” de Hennessy et Patterson, 5ème édition.

Cohérence de la hiérarchie de mémoire partagée

1 Introduction
2 Cohérence des caches
3 Architecture des caches
4 Interface logicielle
5 Optimisations
6 Modèles d'ordonnancement des accès mémoire

Interface logicielle du SE au TLB

- void flush_tlb_all(void), quand la table de page du noyau change car elle est reprise par tous les processus.
- void flush_tlb_mm(struct mm_struct *mm), quand tout l'espace d'un processus change, lors d'un fork ou exec.
- void flush_tlb_range(struct vm_area_struct *vma, unsigned long start, unsigned long end), lorsqu'une partie d'espace d'un processus change, lors d'un munmap.
- void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr), lorsque l'entrée pour une page change, suite à une faute de page.
- void update_mmu_cache(struct vm_area_struct *vma, unsigned long address, pte_t *ptep), indique qu'une nouvelle entrée est disponible pour une page virtuelle, par exemple pour précharger le TLB suite à une faute de page.

Interface logicielle du SE à la cache

- void flush_cache_mm(struct mm_struct *mm), lorsque tout l'espace d'un processus change, lors d'un exit ou exec.
- void flush_cache_dup_mm(struct mm_struct *mm), lorsque tout l'espace d'un processus change, lors d'un fork.
- void flush_cache_range(struct vm_area_struct *vma, unsigned long start, unsigned long end), lorsqu'une partie de l'espace d'un processus change, lors d'un munmap.
- void flush_cache_page(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn), lorsqu'une page change, lors d'une faute de page.
- void flush_cache_kmaps(void), lors d'un changement à highmem.
- void flush_cache_vmap(unsigned long start, unsigned long end), et void flush_cache_vunmap(unsigned long start, unsigned long end), lorsqu'une partie de l'espace virtuel du noyau change, lors d'un map_vm_area ou unmap_kernel_range.

Interface logicielle du SE à la cache

- copy_user_page(...), copy_to_user_page(...), clear_user_page(...), copy_from_user_page(...), faire une copie en tenant compte des problèmes d'alias (pour les dcache indexées virtuellement).
- void flush_dcache_page(struct page *page), lorsque le noyau veut écrire ou lire une page qui peut aussi exister dans l'espace virtuel d'un processus et faire un alias.
- void flush_anon_page(struct vm_area_struct *vma, struct page *page, unsigned long vmaddr), avant que le noyau accède une page anonyme qui pourrait être en alias.
- void flush_kernel_dcache_page(struct page *page), après que le noyau ait modifié une page qui peut être en alias, pour enlever la copie de l'espace noyau en cache.
- void flush_icache_range(unsigned long start, unsigned long end), lorsque du code exécutable vient d'être écrit par le noyau car les caches instructions ne s'attendent normalement pas à du code modifiable.
- void flush_kernel_vmap_range(void *vaddr, int size), avant de faire une opération d'E/S sur une région en mémoire, pour propager toute modification en mémoire centrale et enlever de la cache en vue de faire un DMA.
- void invalidate_kernel_vmap_range(void *vaddr, int size), enlever de la cache avant de faire une lecture d'E/S.

Fautes de cache en SMP

- Inévitable (premier accès).
- Conflit (deux blocs chauds sont dans le même ensemble).
- Capacité (le bloc requis a été évincé faute de place).
- Contention pour l'accès en écriture d'une variable.
- Faux partage, deux variables modifiées par des processeurs différents sont dans le même bloc de cache.

Cohérence de la hiérarchie de mémoire partagée

1. Introduction
2. Cohérence des caches
3. Architecture des caches
4. Interface logicielle
5. Optimisations
6. Modèles d'ordonnancement des accès mémoire

Optimisations

- Pipeline superscalaire permettant d'exécuter plusieurs instructions en même temps et dans le désordre, tout en respectant les dépendances.
- Queues d'écriture.
- Queues de messages d'invalidation.
- Bancs de mémoire cache qui peuvent être accédés en parallèle.

Exécution dans le désordre

- Le compilateur optimiseur peut effectuer tout changement qui ne modifie pas le résultat final, par exemple réordonnancer les écritures et lectures (faites dans des variables séparées ne présentant pas de dépendance).
- Le pipeline superscalaire peut faire beaucoup des mêmes réordonnancements que le compilateur.

```
ld R1, A
ld R2, B
ld R3, C
... calcul ...
st R4, Résultat1
st R5, Résultat2
```

```
ld R1, A
ld R3, C
ld R2, B
... calcul ...
st R5, Résultat2
st R4, Résultat1
```

Queues d'écriture

- Les écritures vers la mémoire cache peuvent être mises en queue, pas d'attente synchrone requise.
- Le contenu de la queue est consulté lors des accès en lecture pour assurer une cohérence entre cette écriture récente et une lecture qui suit sur le même CPU. Néanmoins, en mémoire centrale, les accès en écriture sont retardés (W->R).
- Il peut y avoir plusieurs queues, une par banc de mémoire cache. Dans ce cas, l'ordre des écritures peut changer entre deux accès si une queue est plus pleine qu'une autre (W->W).

Queues de messages d'invalidation

- Lorsqu'un bloc partagé est modifié en cache, toutes les copies doivent être invalidées.
- Les messages d'invalidation peuvent être mis en queue. Ceci retarde l'effet des écritures sur les autres processeurs (W->R).
- Il peut y avoir plusieurs queues, une par banc de mémoire cache. Dans ce cas, l'ordre d'arrivée des messages d'invalidation peut changer entre deux adresses si une queue est plus pleine qu'une autre. Ceci retarde et possiblement réordonnance l'effet des écritures sur les autres processeurs (W->W).

Cohérence de la hiérarchie de mémoire partagée

1 Introduction

2 Cohérence des caches

3 Architecture des caches

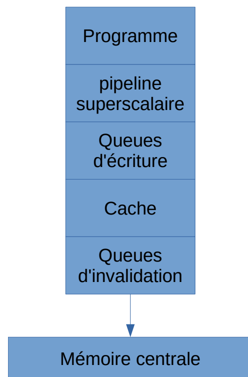
4 Interface logicielle

5 Optimisations

6 Modèles d'ordonnancement des accès mémoire

Contraintes d'ordonnancement

- Ordre séquentiel: aucun réordonnancement (système presque sans optimisation).
- Ordre total des écritures (TSO): les écritures peuvent être retardées par rapport aux lectures (queues d'écriture).
- Ordre partiel des écritures (PSO): les écritures à des variables différentes peuvent être retardées à des degrés divers (queues multiples).
- Ordre faible (WO): les lectures peuvent être retardées (pipeline superscalaire).



Course entre deux processeurs

- La cohérence est maintenue pour un processeur.
- Il n'y a qu'une seule copie maîtresse pour chaque bloc.
- Les mises à jour vers la copie maîtresse peuvent être retardées (queues d'écriture).
- Les mises à jour vers les copies secondaires peuvent être retardées (queues d'invalidation).
- Le code suivant espère que p (next, key, data) sera écrit avant qu'il ne devienne accessible (head.next = p).

```
p->next = head.next;
p->key = key;
p->data = data;

head.next = p;
```

```
p = head.next;
while (p != &head) {
  if (p->key == key) return (p);
  p = p->next;
}
```

Course entre deux processeurs

- Le compilateur peut réordonnancer les accès non dépendants.
- Le pipeline peut réordonnancer les accès non dépendants.
- Une queue d'écriture peut retarder les écritures.
- Des queues d'écriture peuvent réordonnancer les écritures.
- Une queue d'invalidation pourrait retarder la mise à jour de p et head.
- Des queues d'invalidation pourraient retarder la mise à jour de p (data, key, next) même si head.next pointe déjà à p.

Forcer un ordonnancement

- Instruction spéciale pour le compilateur lui demandant de ne pas réordonnancer les accès (lecture, écriture ou les deux).
- Barrière dans le pipeline empêchant de réordonnancer les accès (lecture, écriture ou les deux).
- Barrière dans les queues d'écriture pour retenir toutes les nouvelles écritures tant que celles avant la barrière ne sont pas terminées.
- Barrière qui, lorsqu'une lecture arrive, la bloque le temps de traiter toutes les invalidations en queue.

Les instructions de barrière

- barrier(): barrière pour le compilateur.
- cmm_smp_rmb() en mode usager (ou smp_rmb() dans le noyau Linux): barrière lecture pour le compilateur et le pipeline, et pour les queues d'invalidation.
- cmm_smp_read_barrier_depends(): barrière pour les queues d'invalidation (les valeurs à synchroniser sont dépendantes et ne seront pas touchées par le compilateur et le pipeline).
- cmm_smp_wmb(): barrière d'écriture pour le compilateur et le pipeline, et pour les queues d'écriture.
- cmm_smp_mb(): barrière pour le compilateur, le pipeline, les queues d'écriture et les queues d'invalidation.

Exemple de problème

- Une barrière cmm_smp_wmb() assure que p arrive en mémoire centrale avant d'être accessible (head.next = p).
- Si p et head sont dans des blocs/bancs de cache différents, leurs messages d'invalidation peuvent être dans des queues différentes.
- Si la queue pour p est plus pleine, head peut arriver avant la mise à jour (invalidation) de p et l'ancienne valeur de p dans la cache du processeur 1 pourrait être lue.
- Une barrière cmm_smp_rmb() est trop conservatrice, cmm_smp_read_barrier_depends() est mieux.

```
/* Processeur 0 */
p->next = head.next;
p->key = key;
p->data = data;
cmm_smp_wmb();
head.next = p;
```

```
/* Processeur 1*/
p = head.next;
while (p != &head) {
  /* ajouter barrière ici! */
  if (p->key == key) {
    return (p);
  }
  p = p->next;
}
```