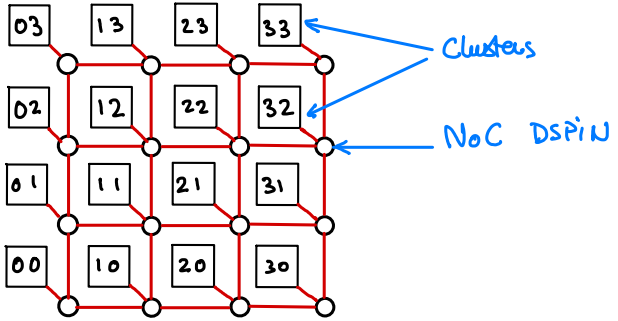
Tsar

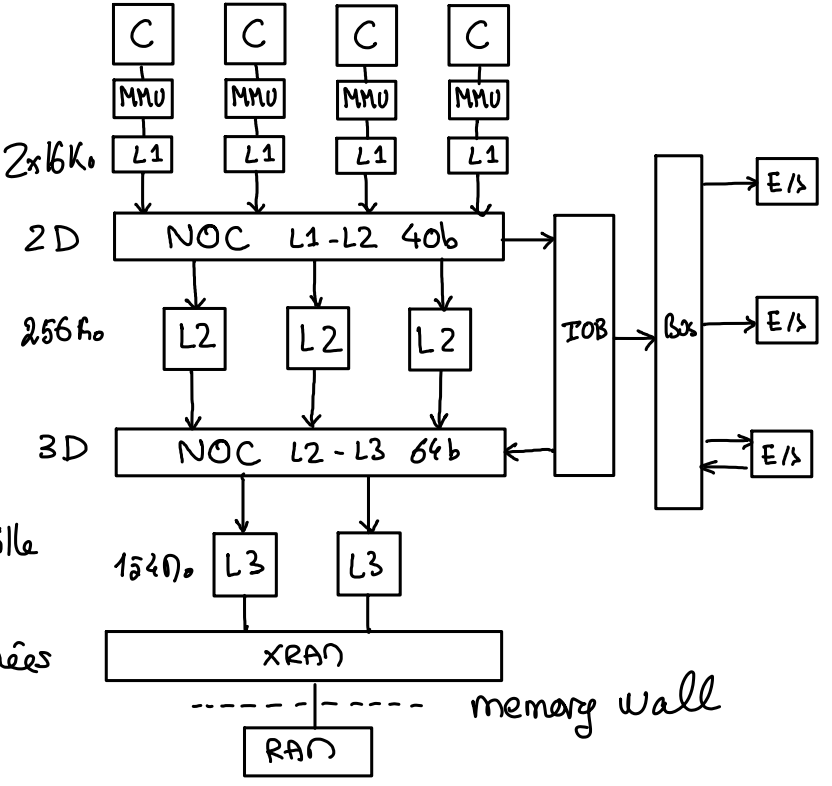
TSAR

- TSAR signifie Tera Scale ARchitecture
- Architecture conçue au LIP6 et dont un prototype a été réalisé au CEA LETI
- 2 projets Européen + 1 projet ANR entre 2008 et 2017 (9 thèses + beaucoup de stages)
- Manycore jusqu'à 1024 cores, les prototypes ont 16 cores et 96 cores
- architecture CC-NUMA : Cache Coherent Non Uniform Memory Access
- Usage général donc tournant sous Linux, NetBSD et ALMOS
- Architecture clusterisée : 1 cluster = 4 MIPS + 1 segment de mémoire + périphériques.
- 3 niveaux de caches 2x L1 de 16kB/core + 1 L2 de 256kB/cluster + 1 L3 de 4 Mo/cluster
- MMU entièrement hardware (pages faults gérées par le matériel)
- cache L1 et TLB cohérents par le matériel

General Purpose CC-NUMA

Les NoC sont des Mesh 2D ou 3D





parce que les Bus ont une bande passante limitée

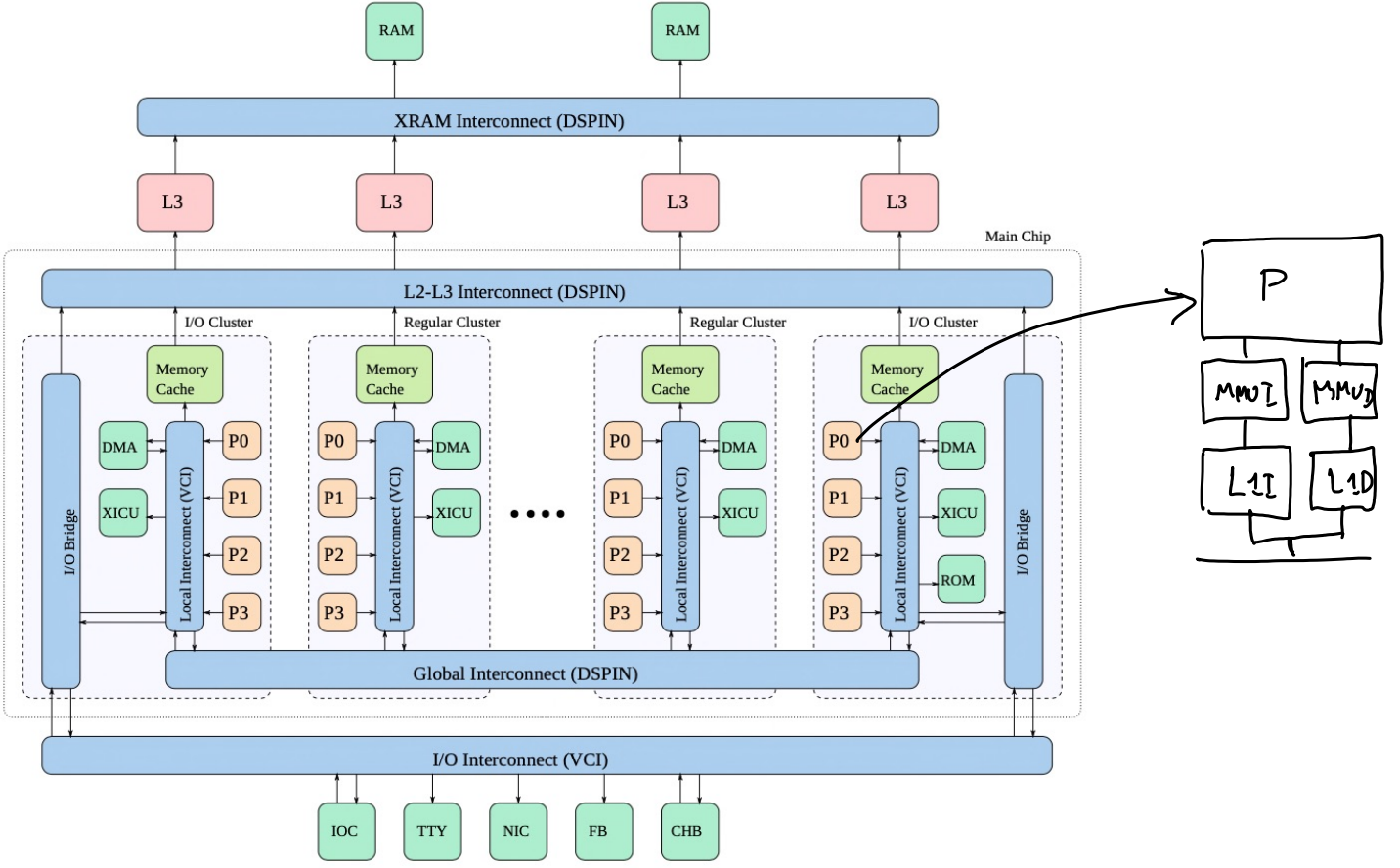
Les NOC ont une bande passante qui croît avec leur taille

MAIS

- latence plus grande ⇒ transactions pipelinées
- cohérence par SNOOP impossible
- problème des instructions atomiques
- l'accès aux E/S est plus complexe

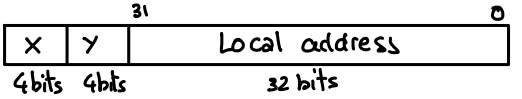
La complexité de TSAR est dans son système-mémoire

General Purpose CC-NUMA

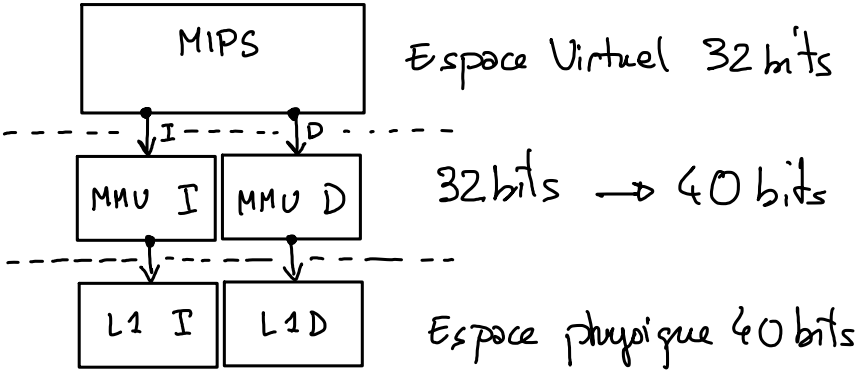


Espaces d'adressages

- Espace d'adressage physique sur 40 bits
  Logiquement partagée => ts les cores y accèdent
  physiquement distribuée => chaque duster gère 1 segment de 4 GB.



- Espace d'adressage virtuel sur 32 bits (parce que les cores sur 32 bits.)



Espace Virtuel 32 bits

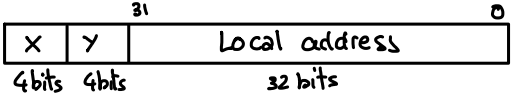
32 bits → 40 bits

Espace physique 40 bits

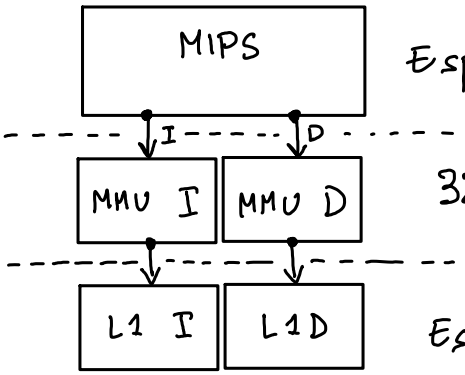
en général c'est l'inverse
l'espace virtuel est plus grand
(ou égal) que l'espace physique

Memory Management Unit

- Espace d'adressage physique sur 40 bits
  logiquement partagée => ts les cores y accèdent
  physiquement distribuée => chaque cluster gère 1 segment de 4 GB.



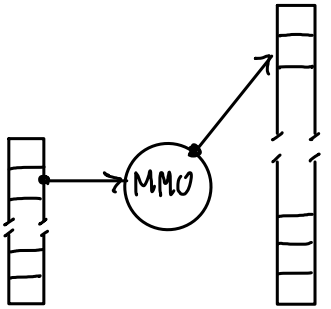
- Espace d'adressage virtuel sur 32 bits (parce que les cores sur 32 bits)
- les Espaces sont de tableaux de pages de 4 KB (2^{12} octets)



Espace Virtuel 32 bits

32 bits → 40 bits

Espace physique 40 bits



espace virtuel 2^{20} pages

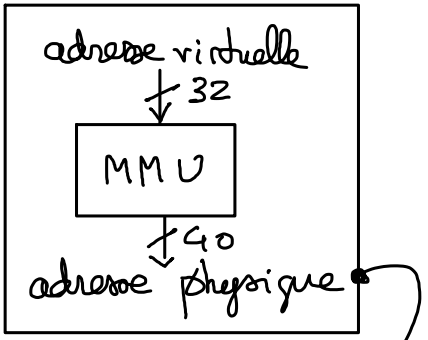
espace physique 2^{28} pages

256 fois plus grand

Memory Management Unit

PTE : page table entry

PTD : page table directory



MMU principe

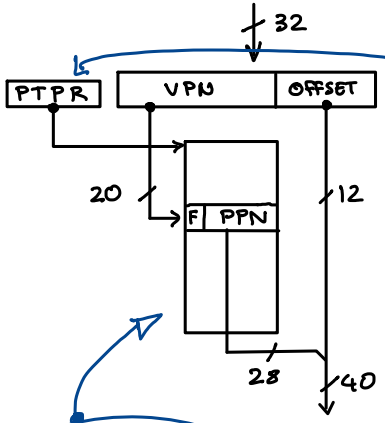


table des pages
$4MB = 2^{20}$ entrées de 32bits minimum

MMU réelle

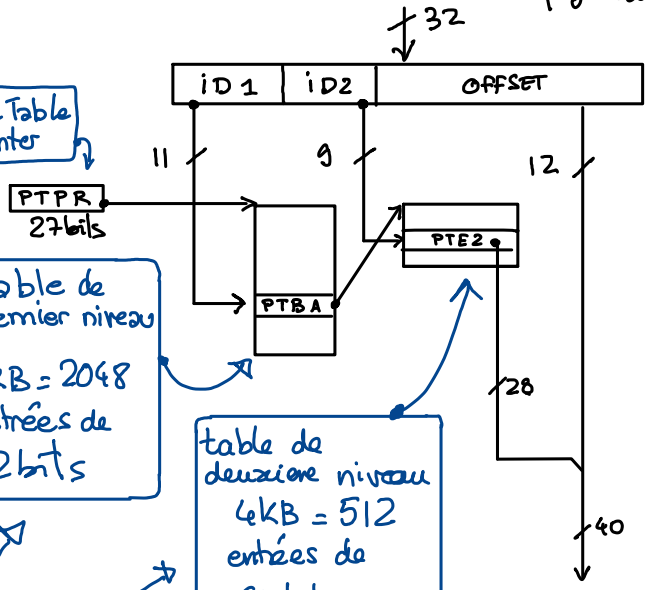
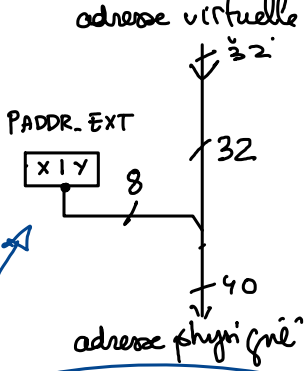


table de premier niveau
8KB = 2048 entrées de 32bits

table de deuxième niveau
4KB = 512 entrées de 64 bits

La table de pages est composée de
- 1 table de premier niveau
- Ø à 2048 table de 2nd niveau.

MMU par défaut



au reset PADDR_EXT = Ø
⇒ ts les clusters accèdent au banc ØØ

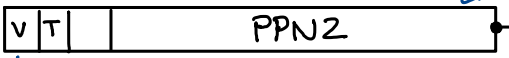
Il y a 2 MMU dans TSAR

Memory Management Unit

La table de premier niveau contient 2 types d'entrée

- PTD1 Page Table Directory.

PPN : Physical Page Number



→ vers table de 2ème niveau

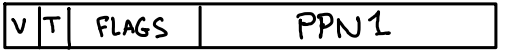
28 bits

validité

type

c'est une adresse de page de 4kB
il faut décaler de 12 bits
vers la gauche

- PTE1 Page Table Entry n°1

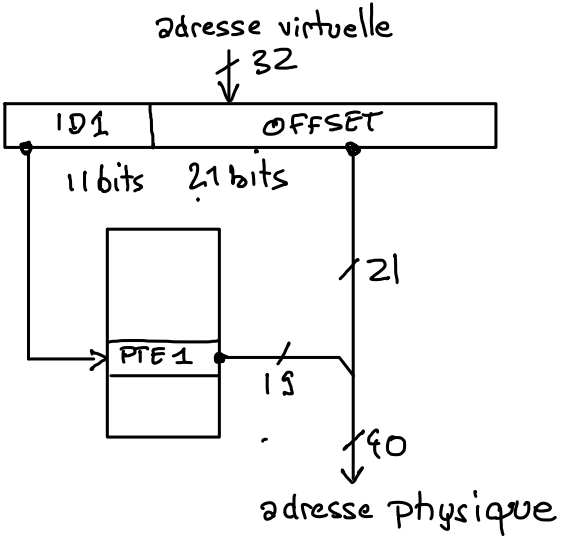


19 bits

c'est une adresse de page de 2MB

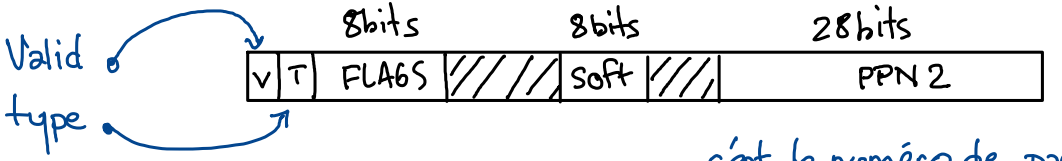
8 FLAGS
- C : cachable
- E : Executable
- W : Writable
- etc...

MMU Grandes Pages



Memory Management Unit

La table de 2nd niveau contient des entrées sur 64 bits.



c'est le numéro de page physique qui doit être concaténé à l'offset

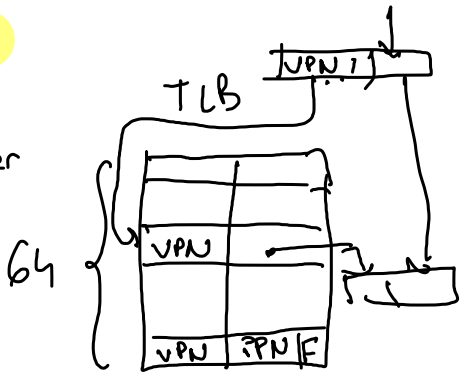
FLAGS

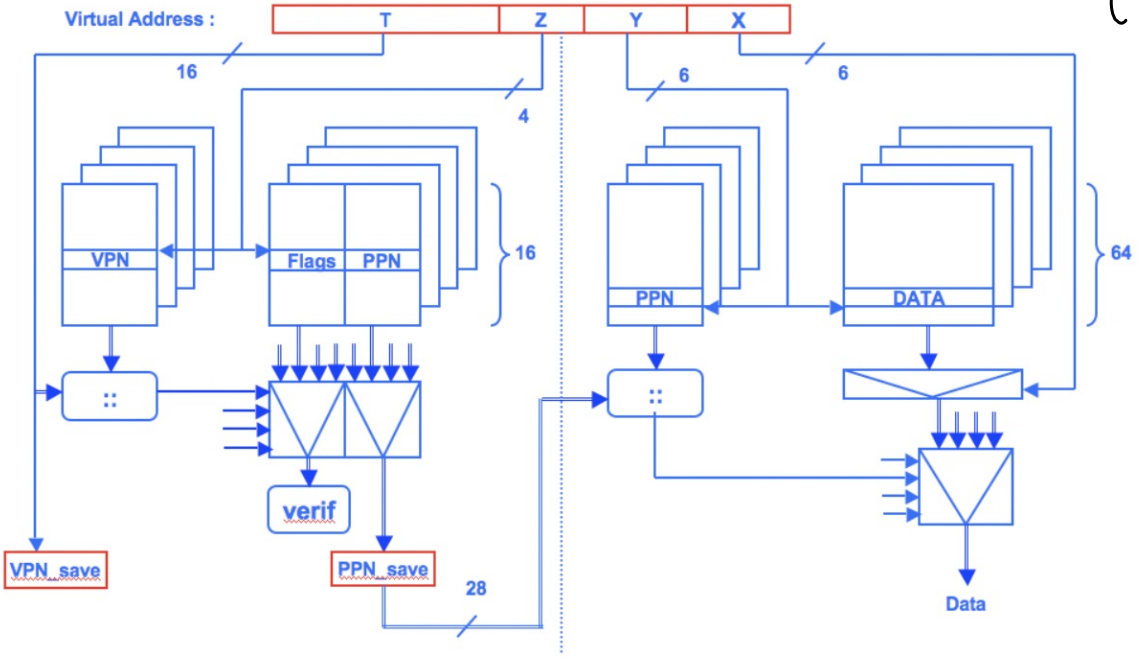
L	Local Acces	hard
R	Remote Access	hard
C	Cachable	soft
W	writable	soft
X	eXecutable	soft
U	User	soft
G	Global	soft
D	dirty	hard

les bits doivent être modifiés par des accès atomiques pour être sûr de leur modification avant de continuer

Memory Management Unit

- Dans la MMU il y a aussi une TLB Translation Lookaside Buffer
  C'est un cache qui enregistre les dernières traductions d'adresses
  il a 64 entrées 16 x 4 (associatif à 4 voies)





Cohérence des caches

La cohérence des caches L1 est garantie par le matériel
Si deux caches L1 lisent la même ligne, alors si l'une des copies est modifiée l'autre aussi
Le snoop ne fonctionne pas avec le NoC,

C'est le cache L2 qui donne les lignes aux caches L1. Il existe plusieurs solutions possibles :

1 - pour chaque ligne, le L2 dispose d'un BIT MAP avec autant de cases que de L1
c'est simple mais ça ne passe pas à l'échelle

2 - les copies de lignes sont chaînées entre elles
c'est très difficile à gérer, quand une ligne est evincée d'un L1 il faut maintenir le chaînage.

3 - le cache L2 se stocke pour chaque ligne la liste des L1 ayant la copie
C'est plus simple à mettre en oeuvre que la solution 2 mais ce n'est pas scalable

4 - le cache L2 se contente de se souvenir le nombre de copies de chaque ligne
C'est simple et ça passe à l'échelle mais il n'est pas possible de faire des mises à jour de ligne
Si une ligne est modifiée - le cache L2 doit demander une invalidation avec 1 broadcast

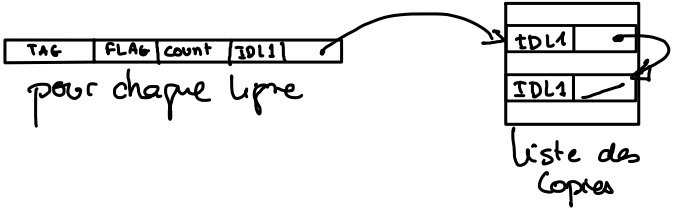
Pour toutes les solutions le cache L2 doit être inclusif et les caches L1 doivent être en write through.

Cohérence des caches

TSAR implémente un mix. entre les solutions 3 et 4 - c'est le protocole **DHCCP**

DHCCP **D**istributed **H**ybrid **C**ache **C**oherence **P**roto**c**ol

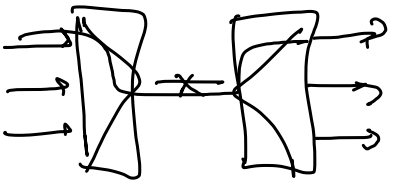
Les caches L2 conservent une liste des copies jusqu'à un seuil (ici 4) ⇒ mise à jour des copies
et au delà de ce seuil il ne stocke que le nombre de copies. ⇒ invalidation des copies

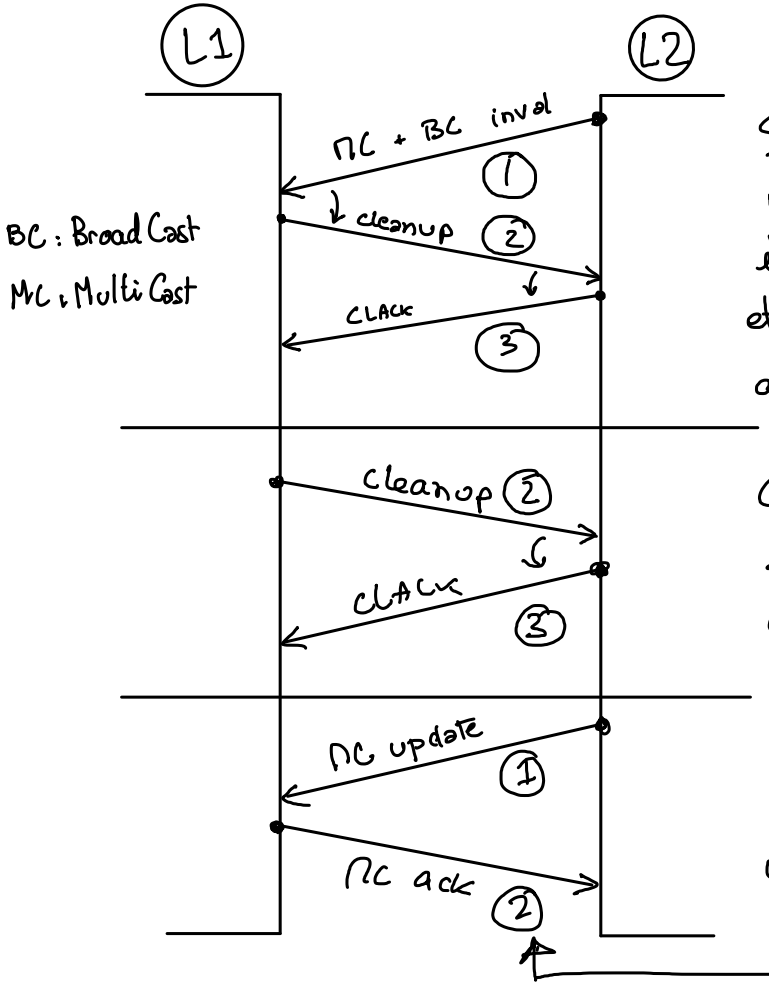


Les transactions entre les caches L1 et L2 sont

- les transactions directes
  - commande pour la lecture ou les écritures
  - réponse pour les lignes lues ou les acquittements d'écriture
- auxquelles s'ajoutent des transactions de cohérences
  - update ou inval L2 → L1 lors de la modification ou l'évincement d'une ligne du L2
  - cleanup L1 → L2 lors de l'évincement d'une ligne du L1
  - clack L2 → L1 pour acquiter le L1 que l'évincement est pris en compte

Cohérence des caches





Si L2 evince des lignes
il doit invalider ces lignes dans tous les L1
il attend autant de cleanup que de copie
et il envoie un clack à la fin
de même s'il reçoit une écriture au delà du seuil

Quand le L1 évince une ligne
il doit prevenir le L2 pour le retirer
de la liste des copies ou décrementer le compteur

Si le L2 reçoit une écriture dans une ligne
en deçà du seuil il demande la mise à jour
individuelle de chaque L1

Il faut 3 réseaux indépendant

Cohérence des caches

- les TLB sont également cohérentes par le matériel
- les entrées de TLB sont aussi dans les L1
- quand une TLB fait Miss ⇒ elle lit le cache L1
  - Si le L1 possède la ligne → Hit
    - la TLB est mise à jour
  - Si le L1 ne possède pas la ligne → Miss
    - lecture de la ligne dans le L1
    - puis mise à jour de la TLB
- le cache sait pour chaque ligne si elle se trouve particulièrement dans la TLB
  - Si une ligne particulièrement dans la TLB est invalidée ou mise à jour la TLB est mise à jour
  - → pb on peut perdre des entrées dans la TLB si le L1 évince des lignes pour se faire de la place

Opérations Atomiques

- Objectif : permettre le partage de ressources entre plusieurs threads, (fil d'exécution)
- mécanisme pour garantir l'atomicité de la séquence : read - modify - write
- MIPS : instructions

Linked load	LL	$r1, ($r2)	$r1 ← mem ($r2)
Store Conditional	SC	$r1, ($r2)	mem($r2) ← $r1 et $r1 ← 1 si OK, 0 si KO

- le cache L2 enregistre qui a réalisé le LL
  et n'autorise le SC que pour celui qui a fait le dernier LL
  la valeur de retour du SC informe si l'écriture a fonctionné

- prendre un lock

```
        la   $r1, spinlock
Lock:   LL   $r2, ($r1)      ] le spinlock est caché
        bnez $r2, lock       ] ⇒ l'attente est dans le cache
        ori  $r2, 1
        SC   $r2, ($r1)
        beqz $r2, lock
```

Opérations Atomiques

- LL addr

  Le L2 dispose d'1 table de réservation associative à 32 entrées : addr, key.

  addr = adresse lue

  key = un identifiant de la transaction sur cette adresse

  Le L2 renvoie la valeur lue et la key.

  Si une autre LL arrive sur la même adresse le L2 renvoie la même chose

- SC addr, val

  Le L1 envoie une commande à l'adresse addr contenant val et Key

  Le L2 recherche l'adresse dans la table de réservation

  s'il y a une entrée et que c'est la bonne key. ⟶ Succès. → écriture et l'entrée est supprimée dans la table de reservation

  Sinon echec

- Problèmes: Il y a un risque de famine pour les LL/SC lointains

  S'il y a un LL sans SC ⟶ la table de réservation se remplie

Opérations Atomiques

Problème 1 : Si LL mais pas de SC
⇒ la table se remplie ⇒ deni de service

Solution : limiter la durée de vie des entrées dans la table de réservation du cache L2

Problème 2. : famine possible pour les LL/SC lointains
à cause des NUMA les LL/SC lointains sont défavorisé

Solution : rendre la durée de vie des entrées variables
beaucoup d'entrée avec une courte durée de vie
et quelques entrée avec une durée de vie grande
Il y a une probabilité non nulle qu'un LL/SC lointain
utilise une entrée longue durée

Conclusion...

la complexité de TSAR est son système mémoire

- Le réseau pour relier le CPU aux mémoires
- les caches et leur cohérence
  (la gestion de la cohérence logicielle des structures de données va s'inspirer de la gestion de la cohérence matérielle)