Soutenance de stage

Augustin LUCAS

Encadré par Guillaume DIDIER, Angeliki KRITIKAKOU
Équipe TARAN, Laboratoire IRISA

Plan

- Contexte
  - Hiérarchie de cache
  - Attaques par canal auxiliaire sur le cache
- Mise en application : analyse d'un système à deux *sockets*
  - Mise en oeuvre
  - Topologie *miss*
  - Topologie *hit* exclusif

Hiérarchie de cache

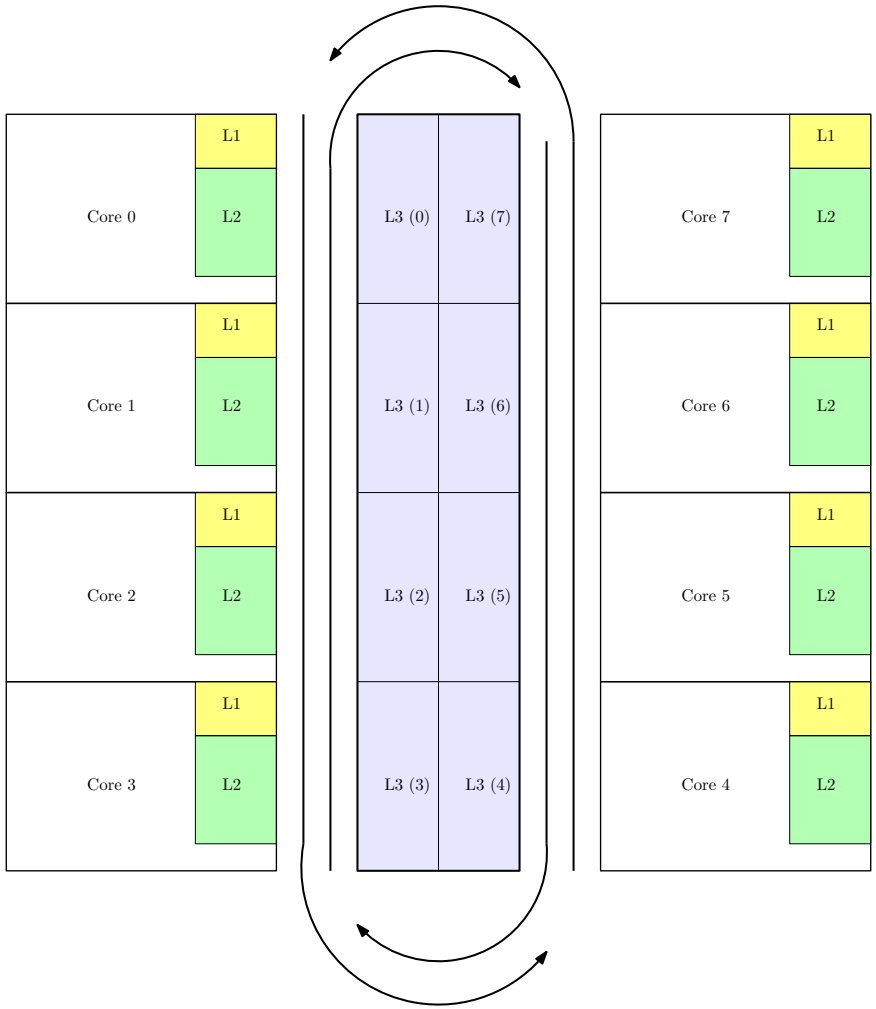
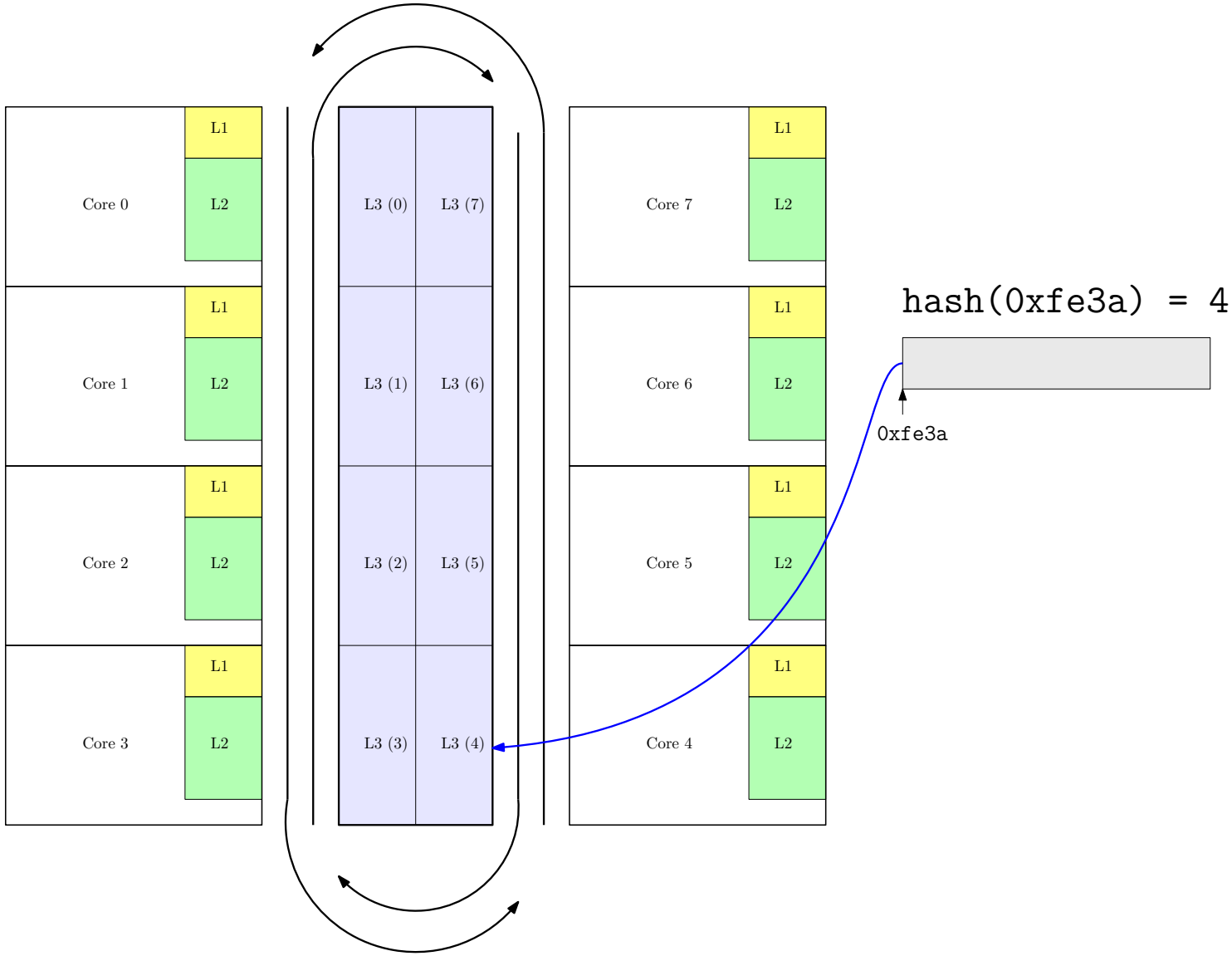


Figure – Topologie d'un processeur *Intel Xeon* jusqu'à *Broadwell* (inclus)

Hiérarchie de cache



Cohérence de cache

Définition (Cohérence de cache)

Faire en sorte que tous les coeurs soient d'accord sur une séquence de valeurs prises par une ligne de cache.

Cohérence de cache

Définition (Cohérence de cache)

Faire en sorte que tous les coeurs soient d'accord sur une séquence de valeurs prises par une ligne de cache.

On définit les états MESI.

Cohérence de cache

Définition (Cohérence de cache)

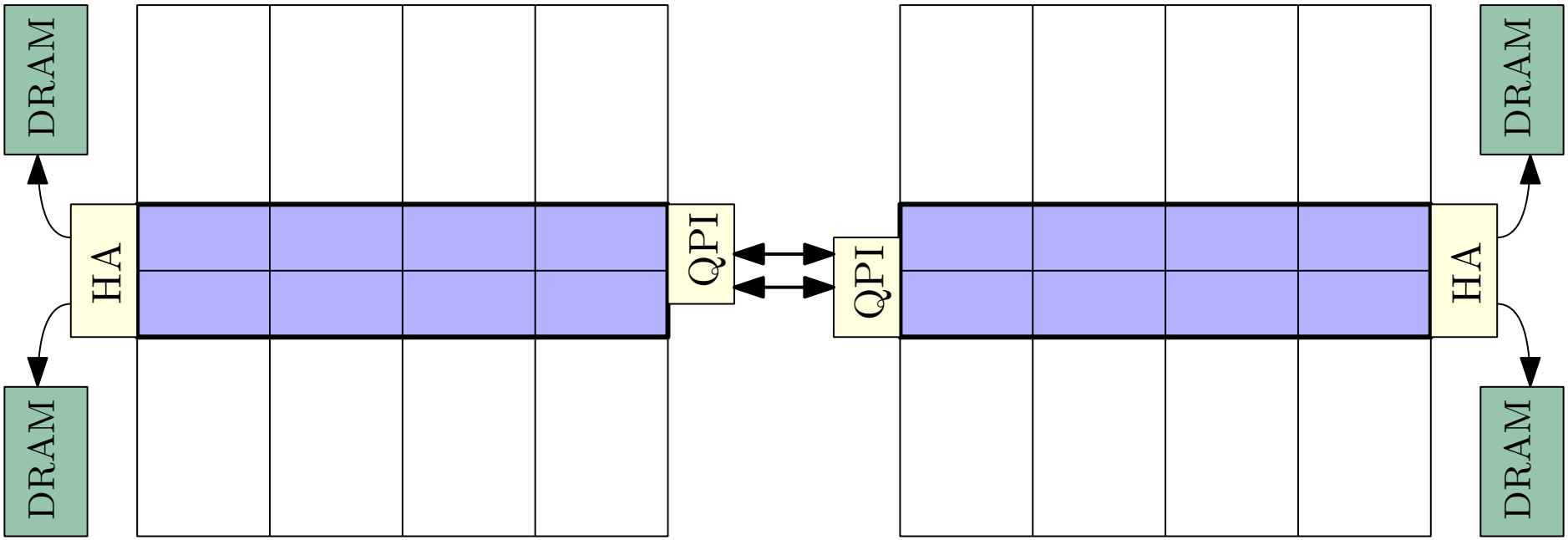
Faire en sorte que tous les coeurs soient d'accord sur une séquence de valeurs prises par une ligne de cache.

On définit les états MESI.

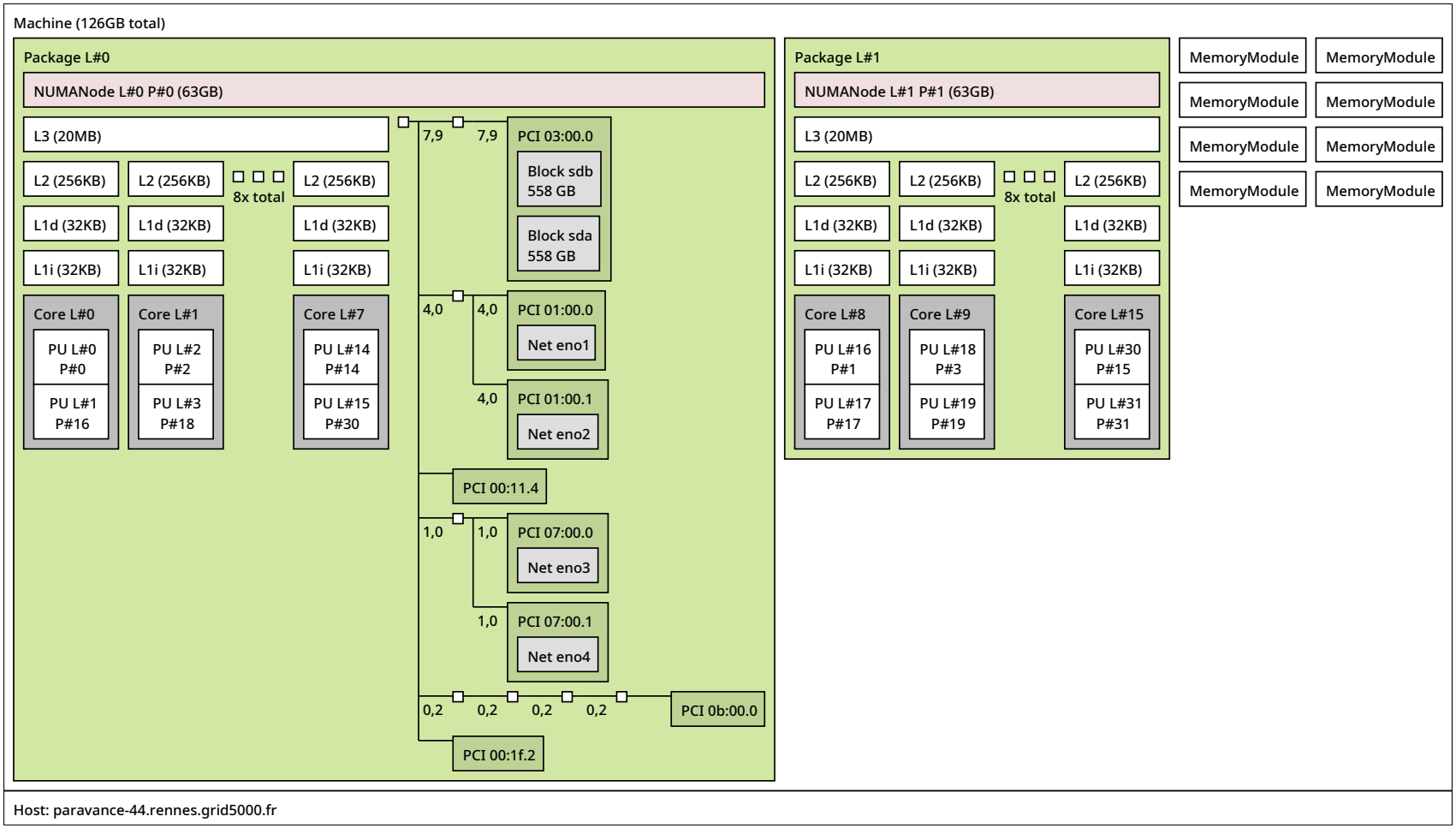
Deux protocoles :

- Solution par annuaire (*directory*)
- Solution par *snooping*

Multi-*socket*



NUMA


Machine (126GB total)
Package L#0
NUMANode L#0 P#0 (63GB)
L3 (20MB)
L2 (256KB)
L2 (256KB)
8x total
L2 (256KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#0
PU L#0 P#0
PU L#1 P#16
Core L#1
PU L#2 P#2
PU L#3 P#18
Core L#7
PU L#14 P#14
PU L#15 P#30
7,9
7,9
PCI 03:00.0
Block sdb 558 GB
Block sda 558 GB
4,0
4,0
PCI 01:00.0
Net eno1
4,0
PCI 01:00.1
Net eno2
PCI 00:11.4
1,0
1,0
PCI 07:00.0
Net eno3
1,0
PCI 07:00.1
Net eno4
0,2
0,2
0,2
0,2
PCI 0b:00.0
PCI 00:1f.2
Package L#1
NUMANode L#1 P#1 (63GB)
L3 (20MB)
L2 (256KB)
L2 (256KB)
8x total
L2 (256KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#8
PU L#16 P#1
PU L#17 P#17
Core L#9
PU L#18 P#3
PU L#19 P#19
Core L#15
PU L#30 P#15
PU L#31 P#31
MemoryModule
MemoryModule
MemoryModule
MemoryModule
MemoryModule
MemoryModule
MemoryModule
MemoryModule
Host: paravance-44.rennes.grid5000.fr

Attaques par canal auxiliaire sur le cache

Objectif des attaques étudiées :

Attaques par canal auxiliaire sur le cache

Objectif des attaques étudiées :
Savoir à quelles adresses mémoire un programme accède

Attaques par canal auxiliaire sur le cache

Objectif des attaques étudiées :
Savoir à quelles adresses mémoire un programme accède

$\longrightarrow$ A permis des attaques récupérant des clés privées via OpenSSL

Attaques par canal auxiliaire sur le cache

Objectif des attaques étudiées :
Savoir à quelles adresses mémoire un programme accède

$\longrightarrow$ A permis des attaques récupérant des clés privées via OpenSSL
$\longrightarrow$ A permis d'implémenter des enregistreurs de frappe (*keylogger*)

clflush

Definition (clflush (d'après le manuel Intel))

CLFLUSH (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor's cache hierarchy, and it is broadcast throughout the cache coherency domain.

clflush

Definition (clflush (d'après le manuel Intel))

CLFLUSH (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor's cache hierarchy, and it is broadcast throughout the cache coherency domain.

- fonctionnement non détaillé par Intel

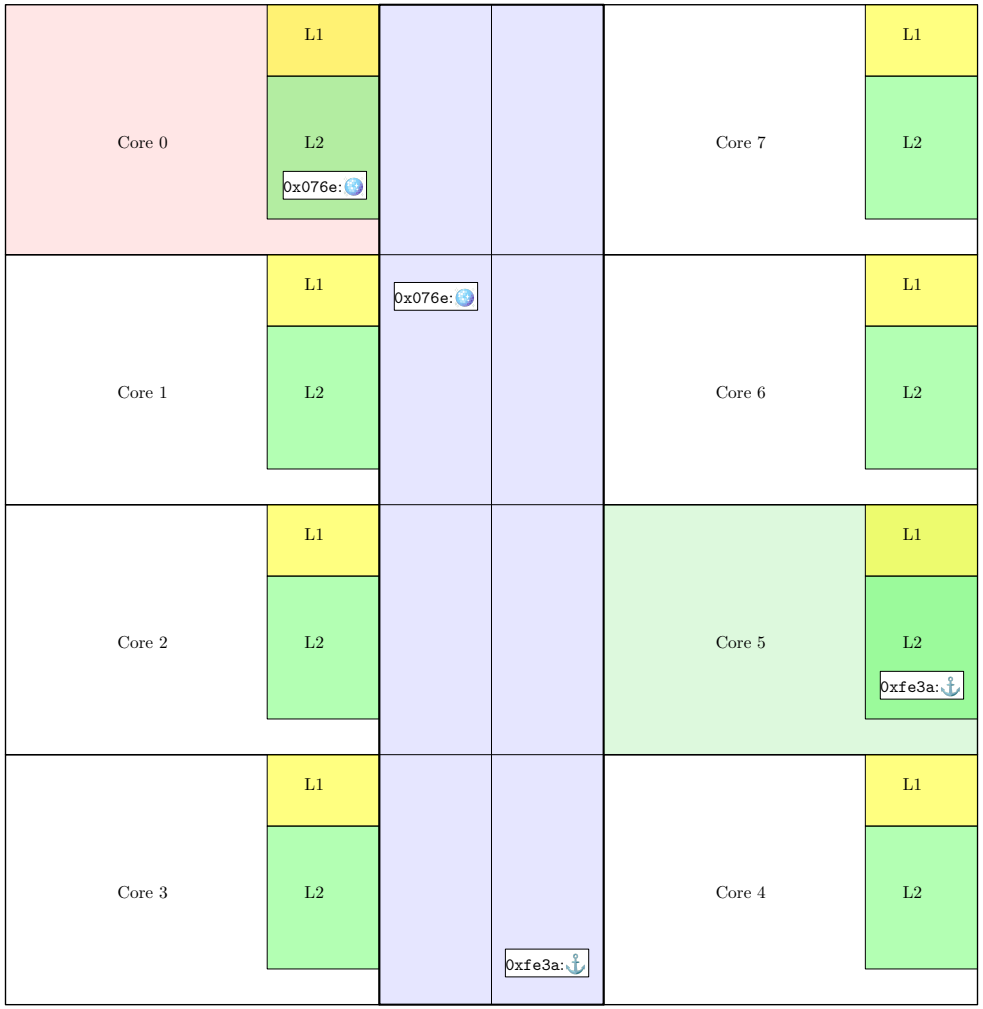
clflush

Definition (clflush (d'après le manuel Intel))

CLFLUSH (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor's cache hierarchy, and it is broadcast throughout the cache coherency domain.

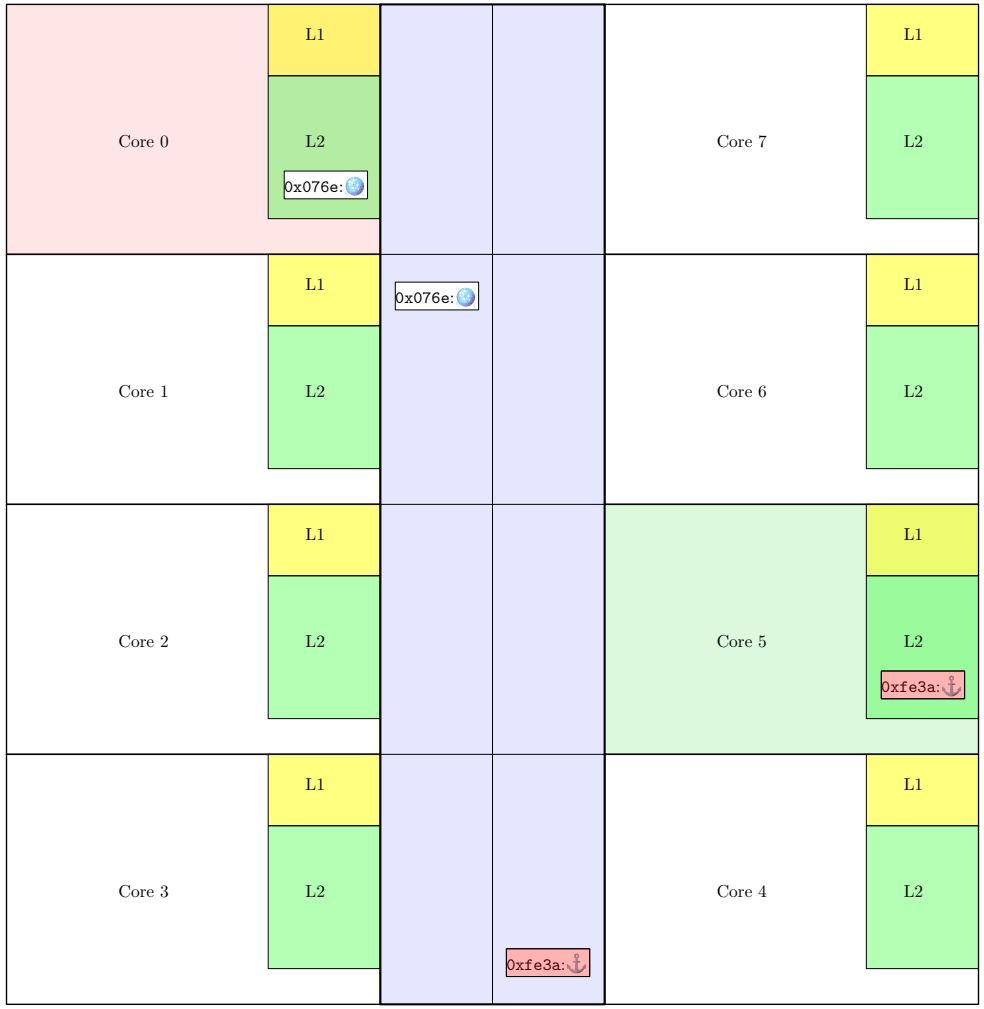
- fonctionnement non détaillé par Intel
- accessible à tout utilisateur non privilégié sur les adresses mémoires auxquelles il a accès.

Flush+Flush



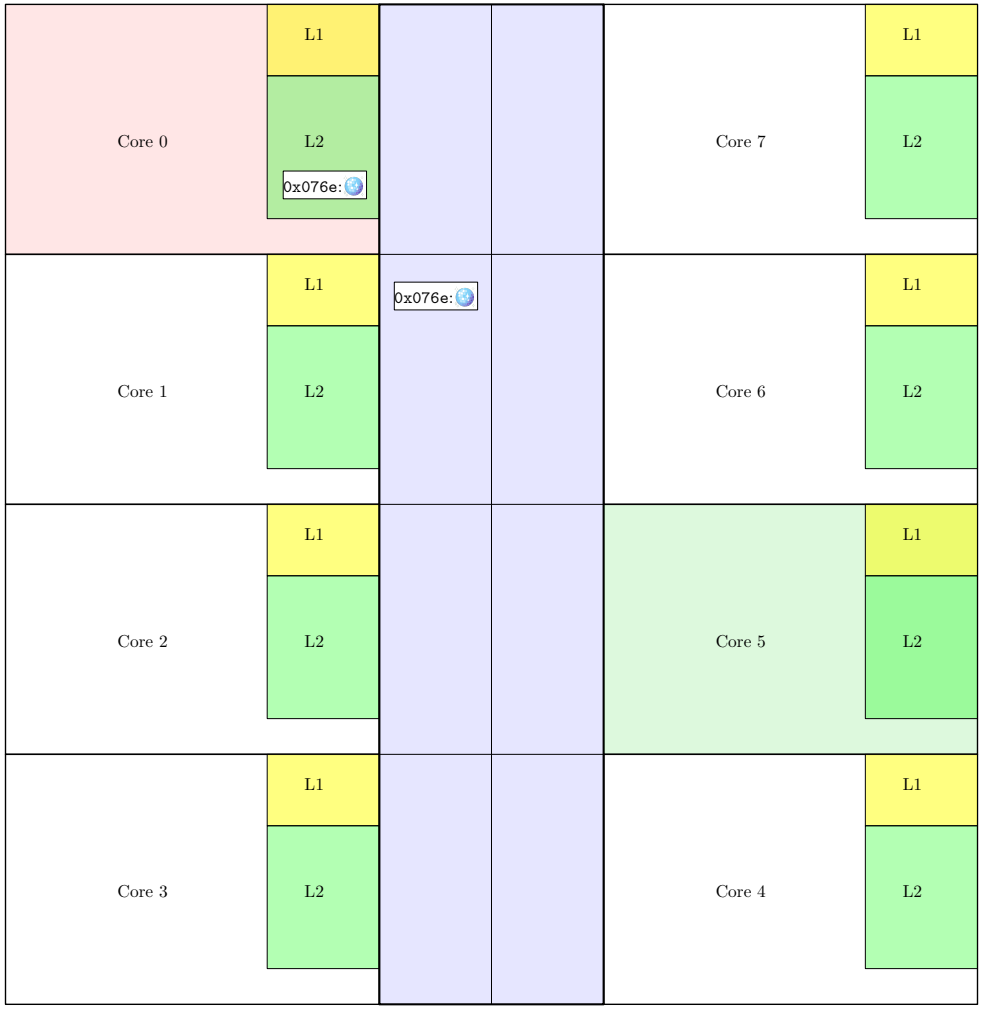
Observons le mécanisme proposé par Flush+Flush

Flush+Flush



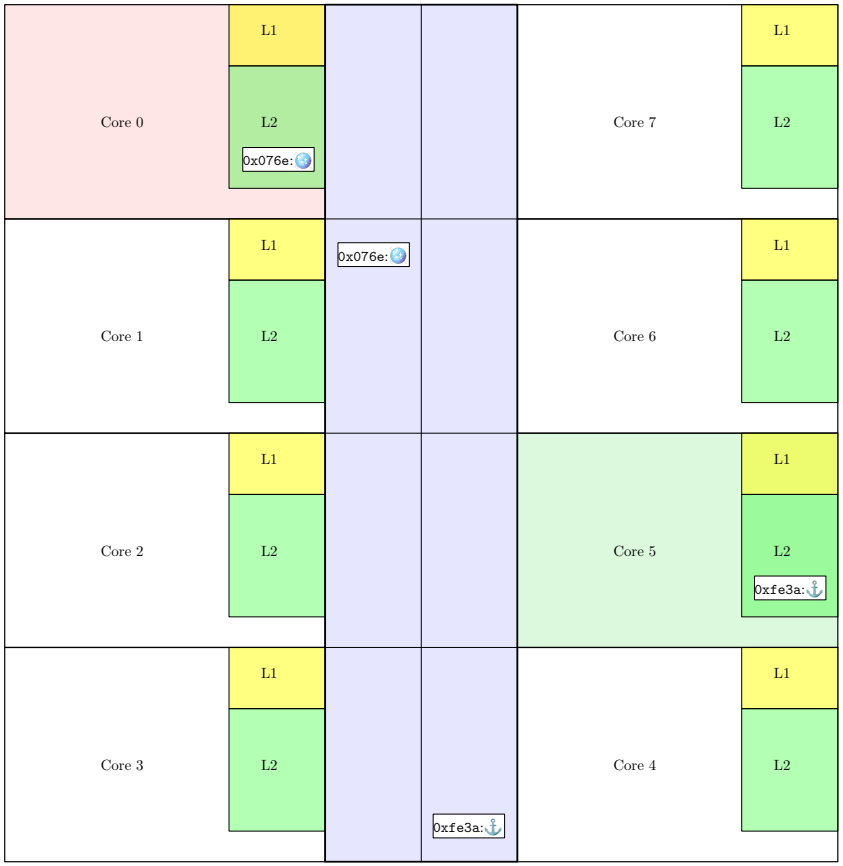
L'attaquant (en *Core 0*) appelle `clflush(0xfe3a)`

Flush+Flush

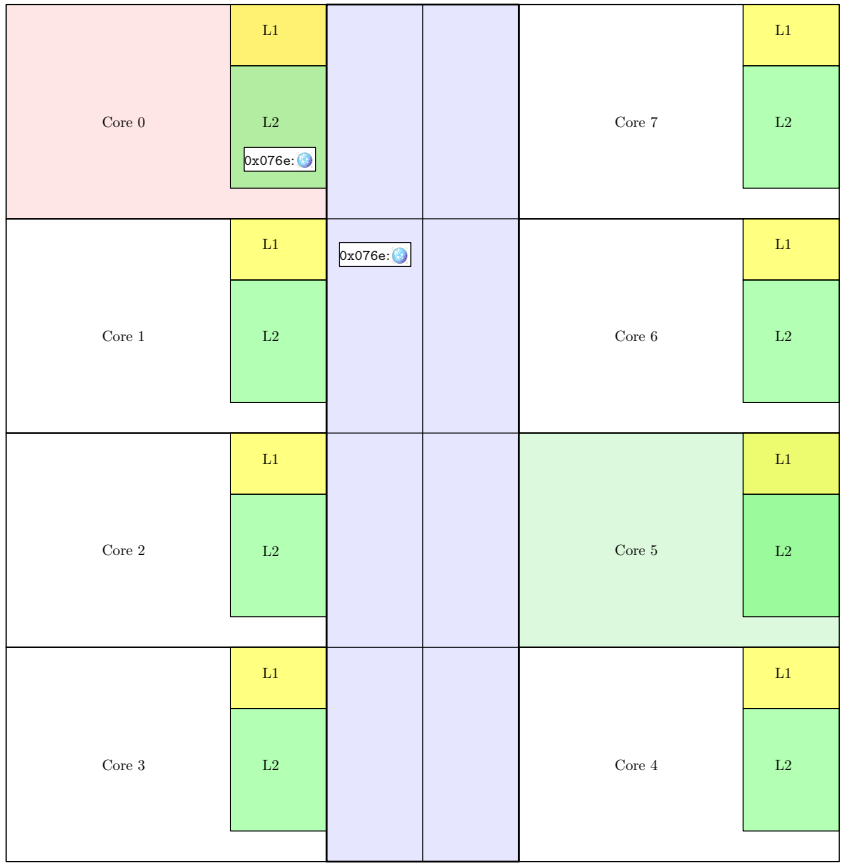


L'attaquant (en *Core 0*) appelle `clflush(0xfe3a)`

Flush+Flush



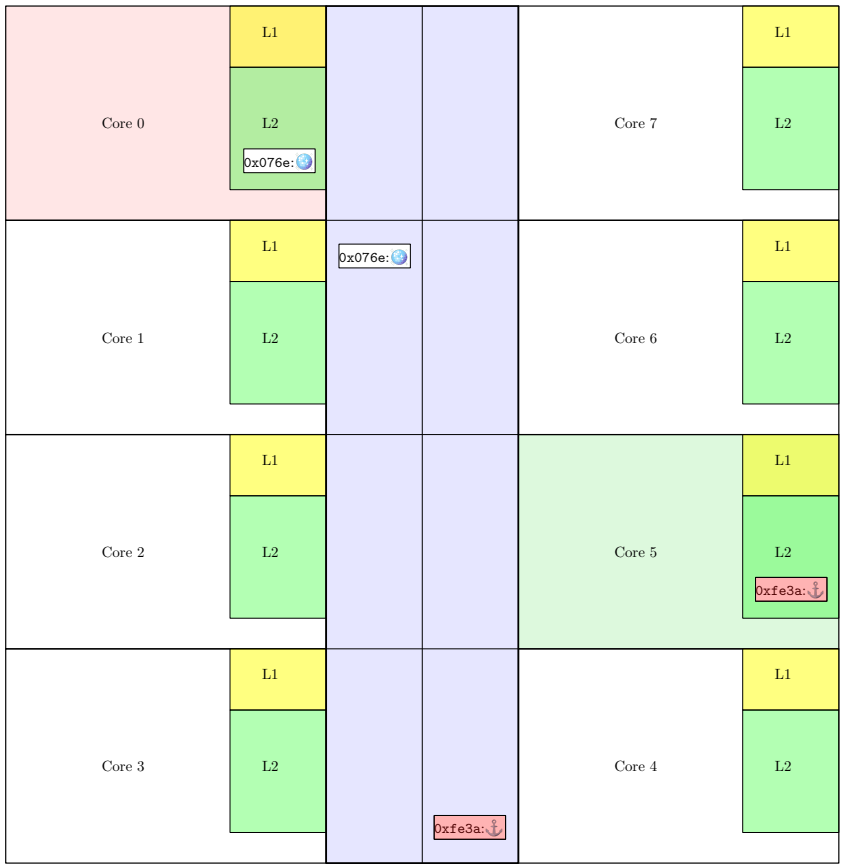
La victime charge `0xfe3a`

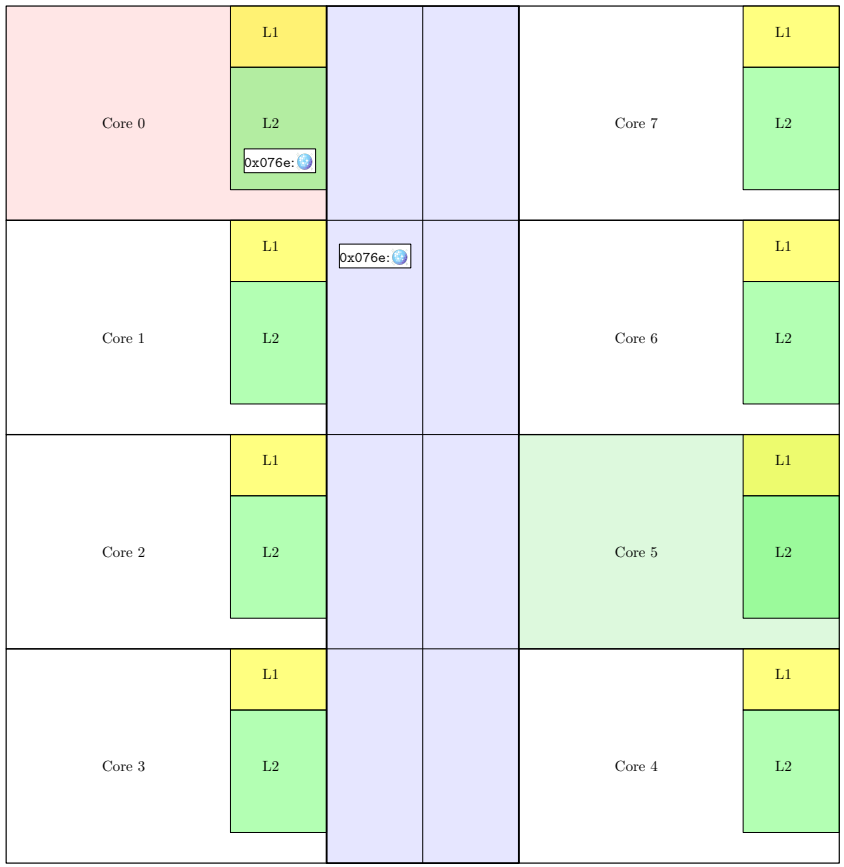


La victime ne fait rien

Flush+Flush

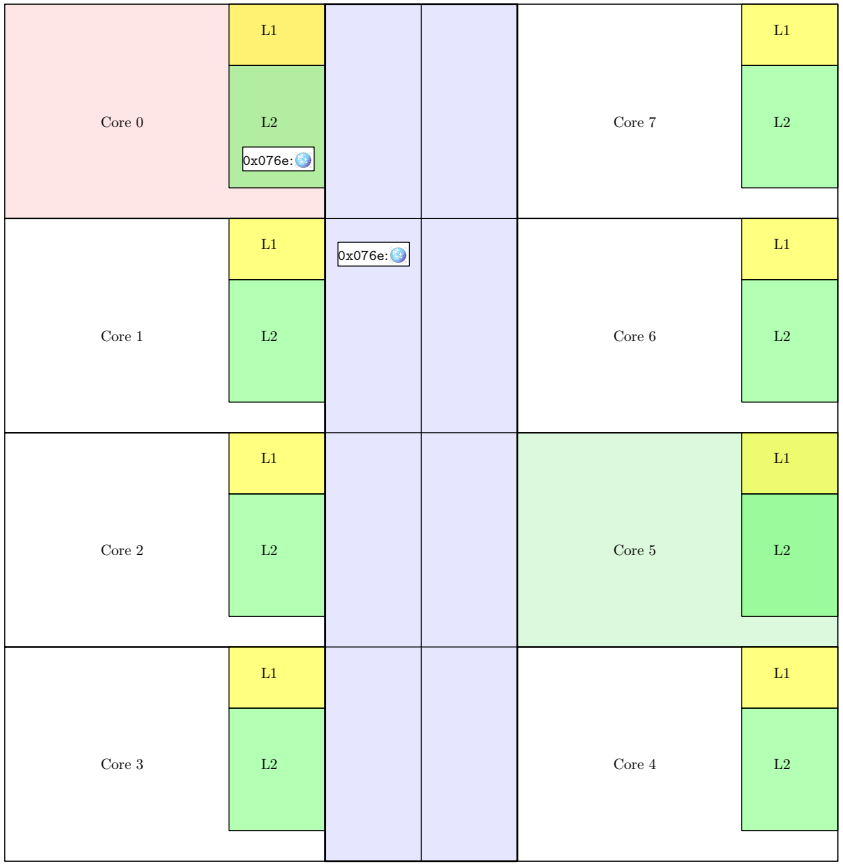
L'attaquant appelle `clflush(0xfe3a)`, en mesurant le temps nécessaire



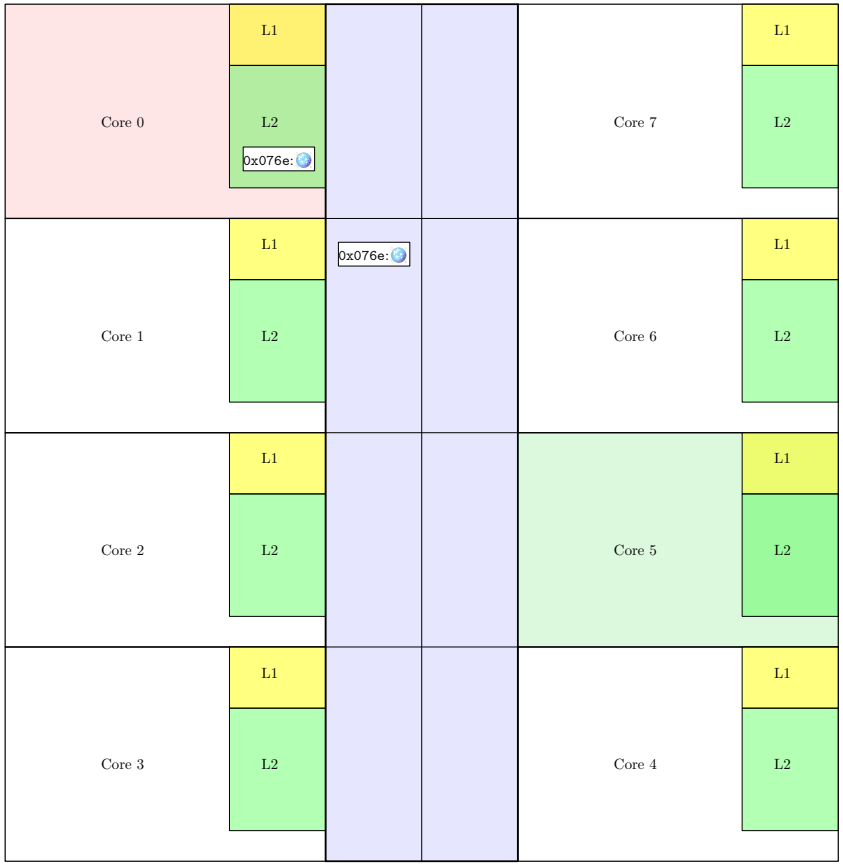


Flush+Flush

L'attaquant appelle `clflush(0xfe3a)`, en mesurant le temps nécessaire



⏱ ~175 cycles



⏱ ~160 cycles

Problématique

- On observe une faible différence de temps

Problématique

- On observe une faible différence de temps
- La décision de *hit* ou *miss* demande une bonne précision

Problématique

- On observe une faible différence de temps
- La décision de *hit* ou *miss* demande une bonne précision
- Le mieux est de calibrer par paire de coeurs, et pour chaque *slice*

Protocole expérimental

Premiers résultats

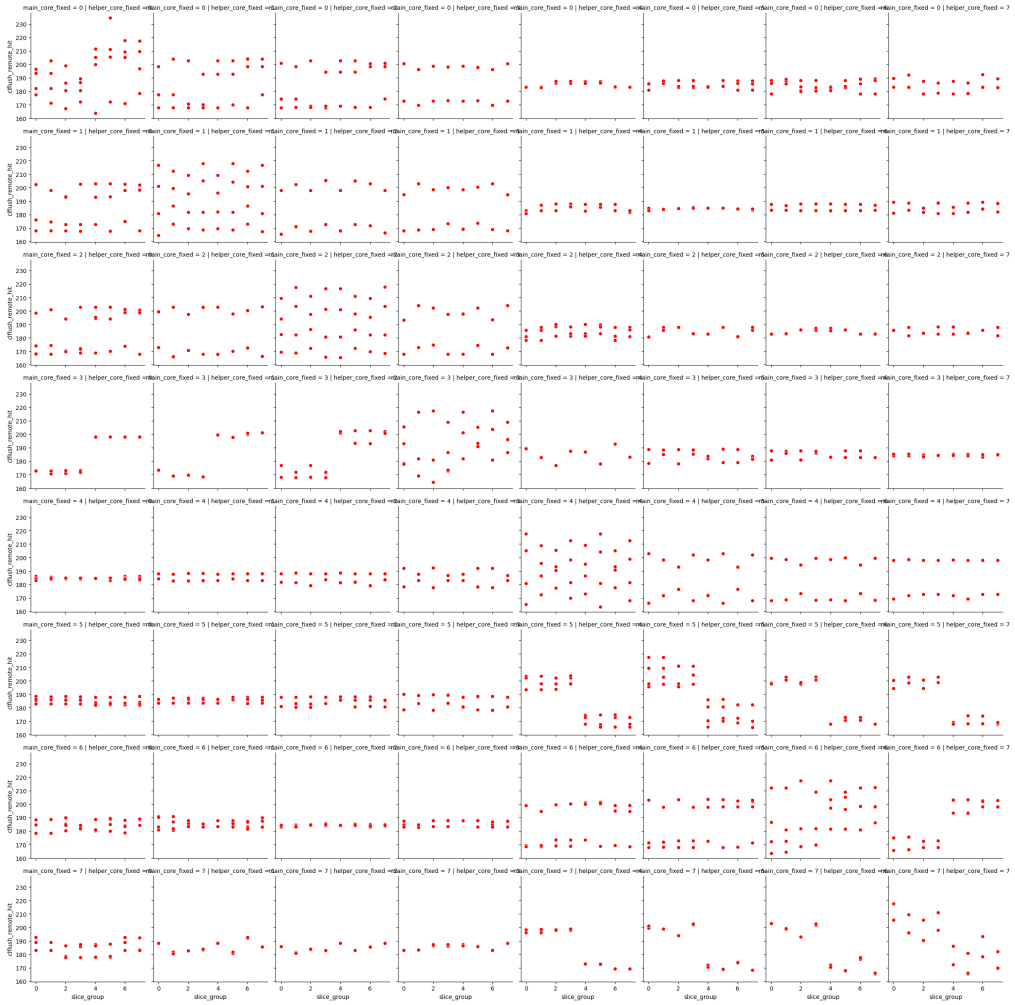


Figure – Premiers résultats sur *paravance* (*hit*)

Premiers résultats

- Beaucoup de bruit

Premiers résultats

- Beaucoup de bruit
- La fréquence, fixée avec le profil **performance** de `cpufreq` ne l'est pas réellement

Premiers résultats

- Beaucoup de bruit
- La fréquence, fixée avec le profil **performance** de `cpufreq` ne l'est pas réellement
- L'architecture NUMA change fréquemment les adresses physiques des données

Résultats après rectification

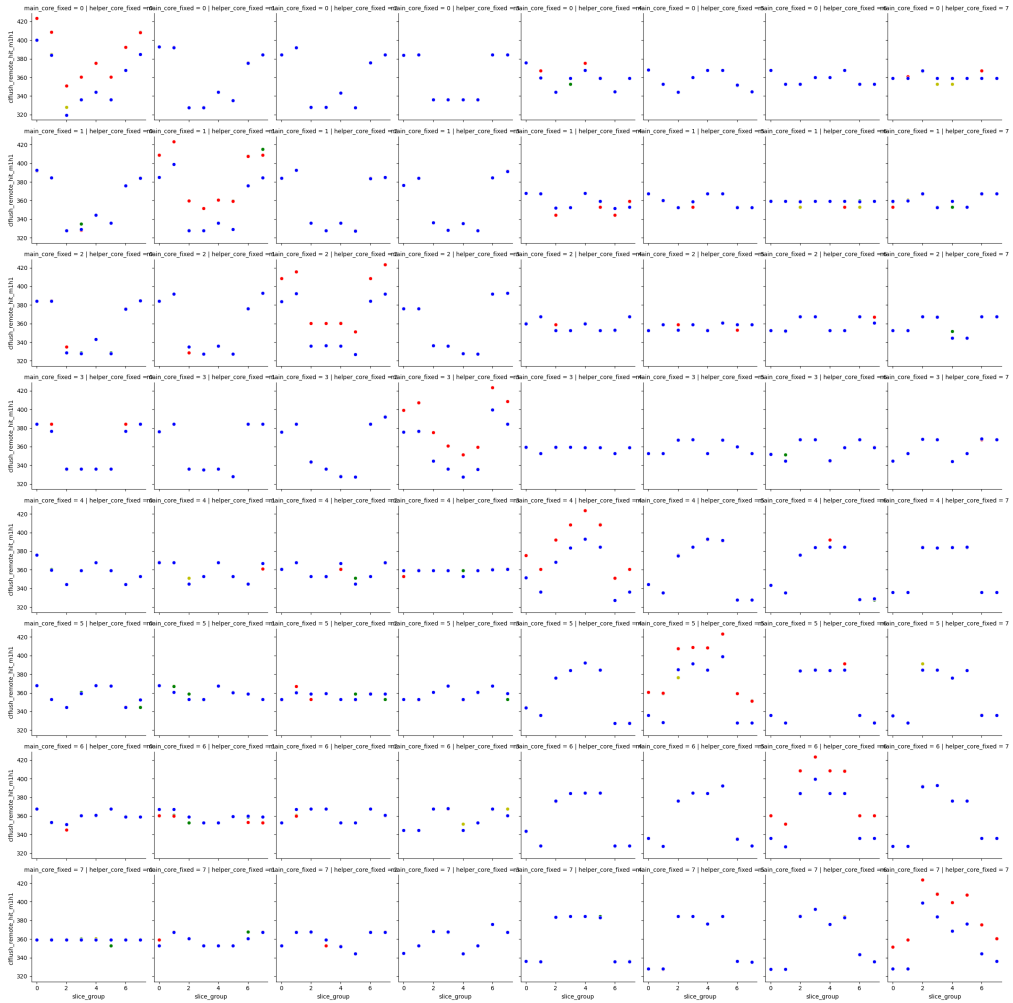
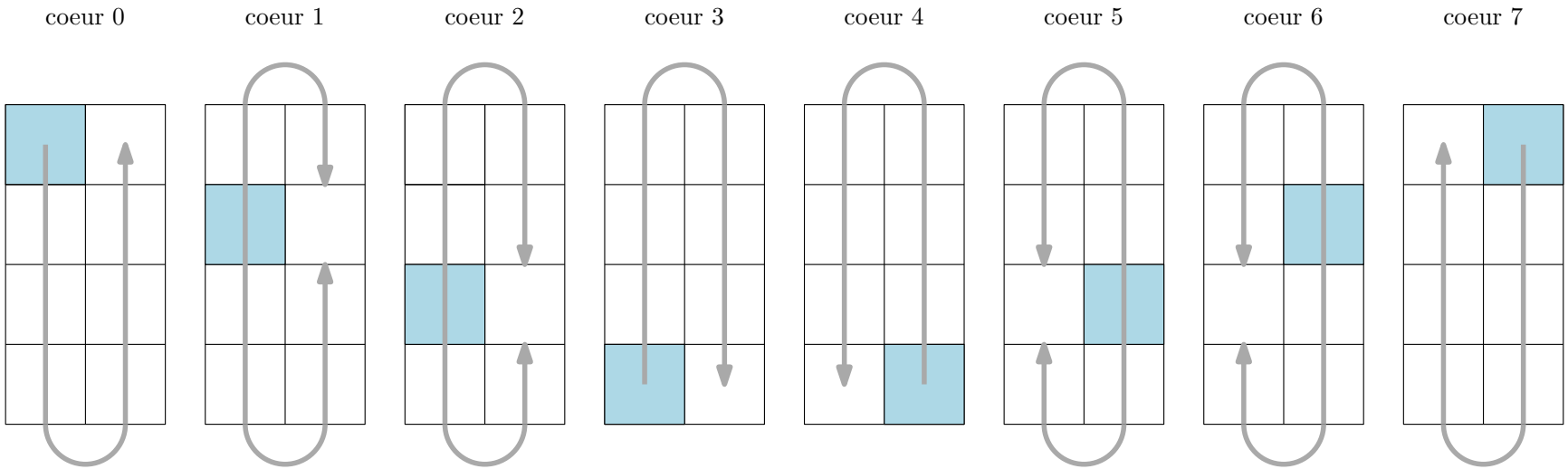


Figure – Nouveaux résultats sur *paravance* (*hit*)

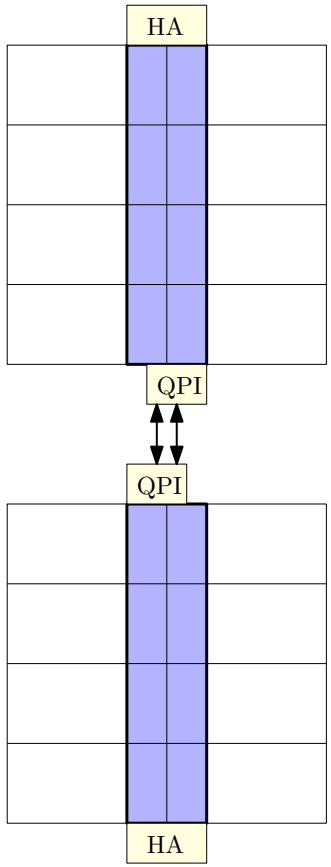
Modèle *miss*

- Ne dépend pas du coeur de la victime
- Le temps minimal à *slice* fixée est croissant selon la distance à l'autre *socket* dans le sens horaire
- Les variations restantes indiquent le chemin suivant



Modèle *miss*

1. Le coeur attaquant contacte la *slice* locale en suivant le chemin précédent
2. La *slice* locale contacte la *slice* distante en passant par le QPI. Le trajet de la *slice* locale au QPI se fait dans le sens horaire, celui du QPI à la *slice* distante dans le sens anti-horaire.
3. Si le *Home Agent* distant doit être contacté, cela se fait à ce moment, en faisant un tour complet de la *socket* pour revenir à la *slice* distante.
4. Le chemin est parcouru à l'envers pour repasser par la *slice* locale jusqu'au coeur attaquant



Modèle *miss*

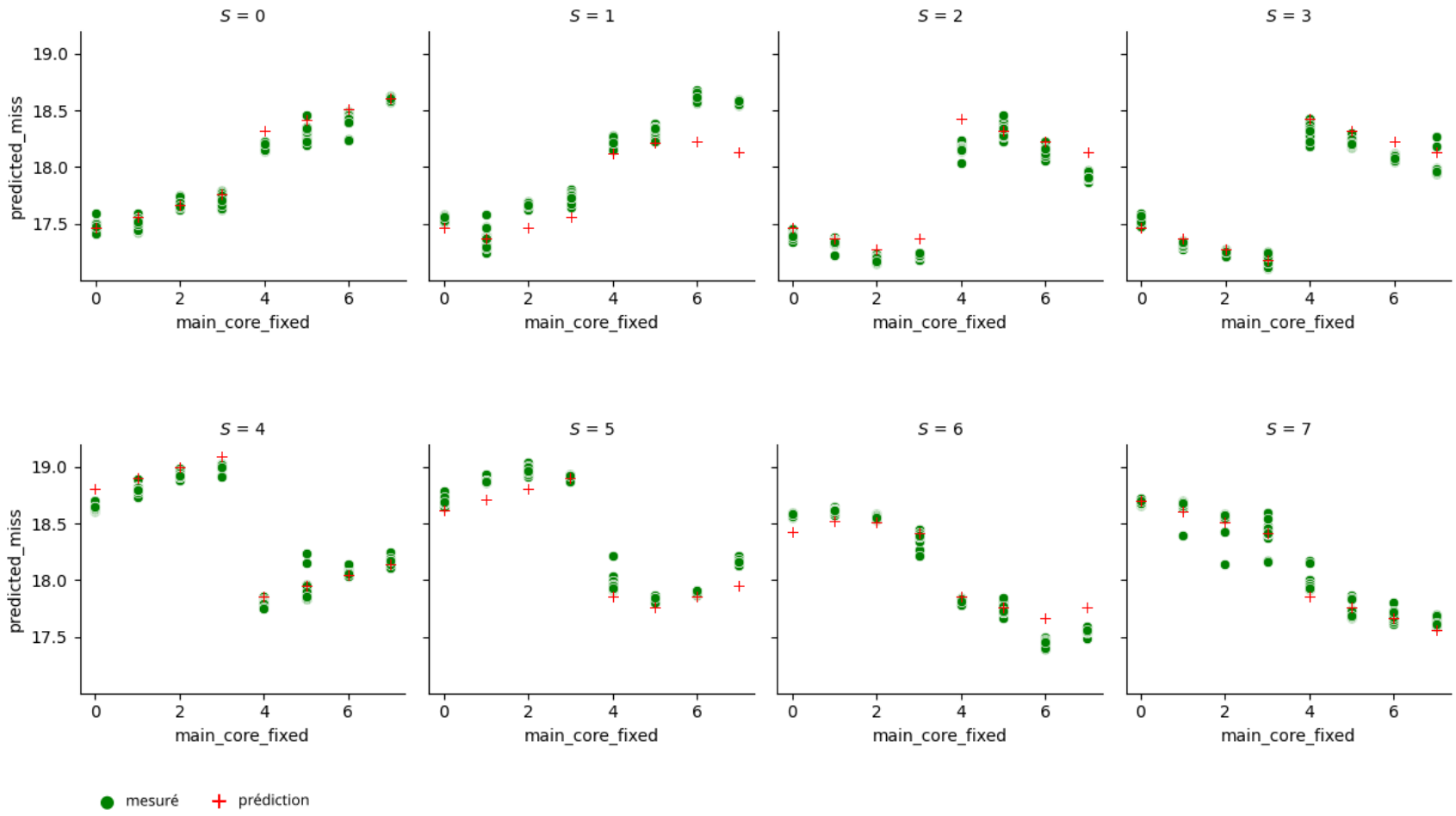


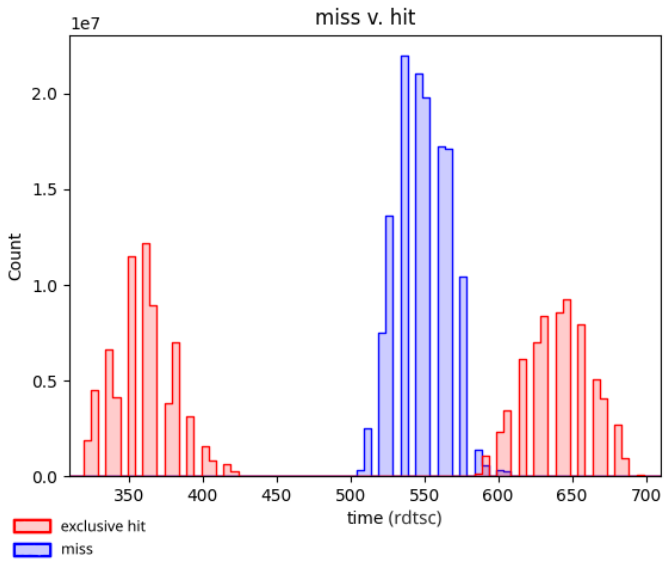
Figure – Prédictions pour un *miss*

Modèle *hit* exclusif

- On cherche un modèle cohérent avec le modèle des *miss*, car le fonctionnement initial doit être le même ;

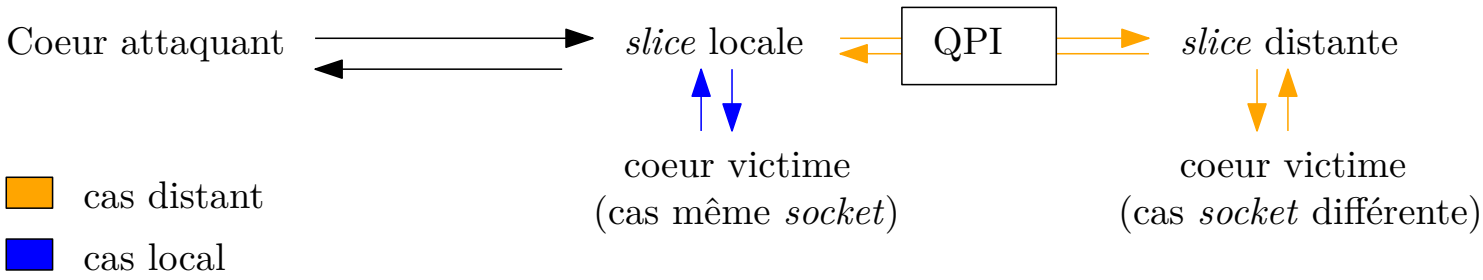
Modèle *hit* exclusif

- On cherche un modèle cohérent avec le modèle des *miss*, car le fonctionnement initial doit être le même ;
- Le temps d'exécution pour un *hit* dépend largement du fait d'être inter-*socket*, donc si la première *slice* contactée contient la ligne en cache, tout semble s'arrêter.



Modèle *hit* exclusif

Cela suggère le modèle suivant :



Modèle *hit* exclusif

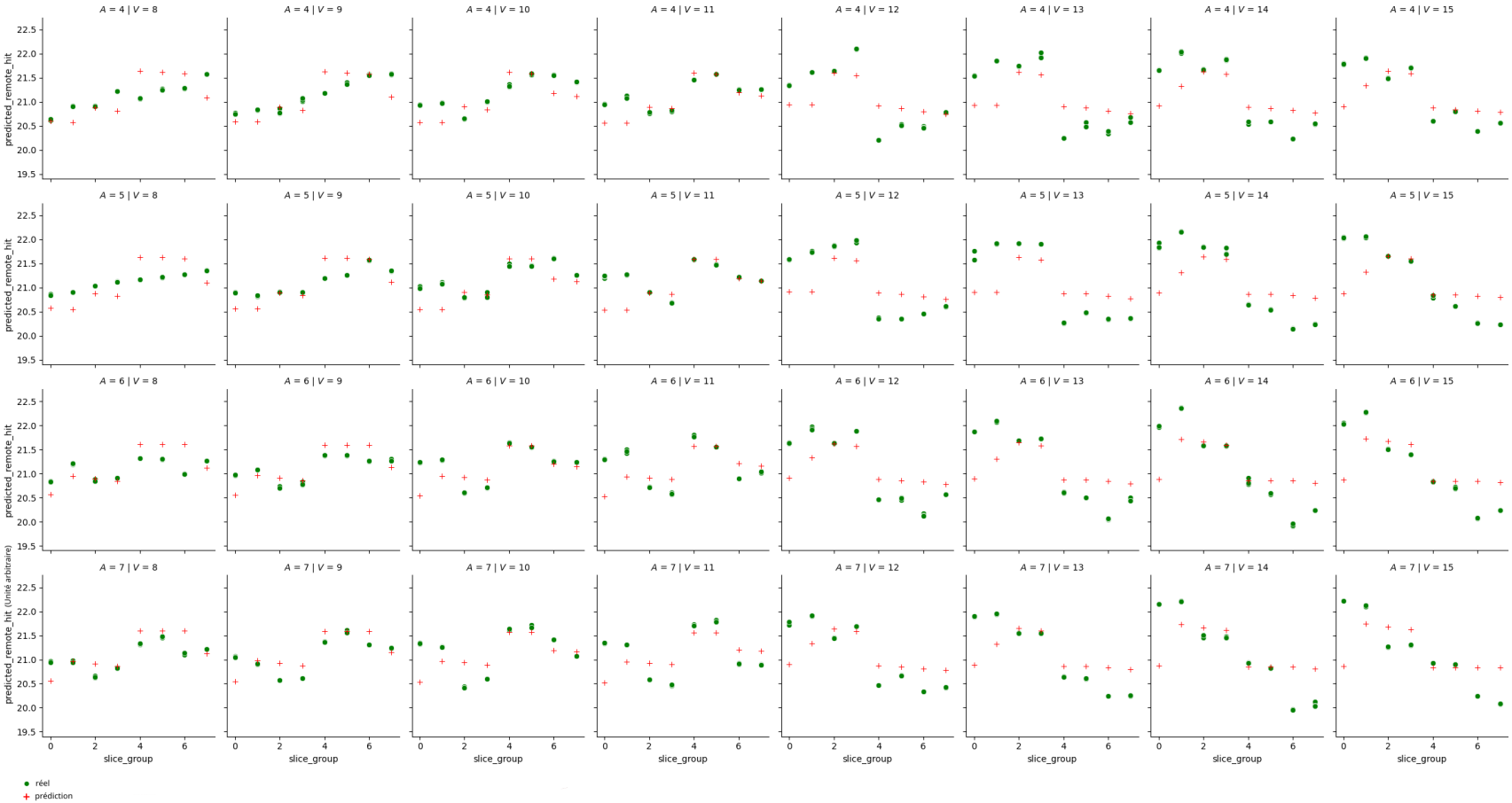


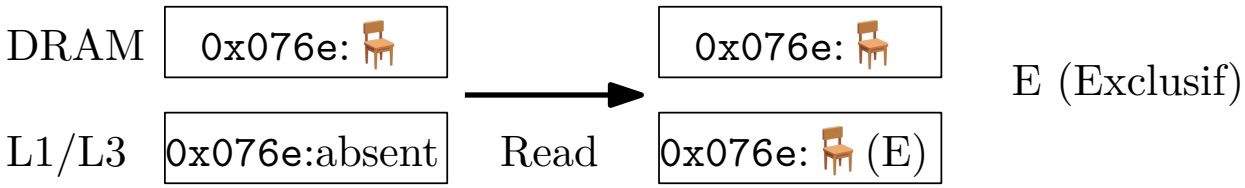
Figure – Prédictions pour un *hit* exclusif au sein de *sockets* différents

Pistes pour la suite

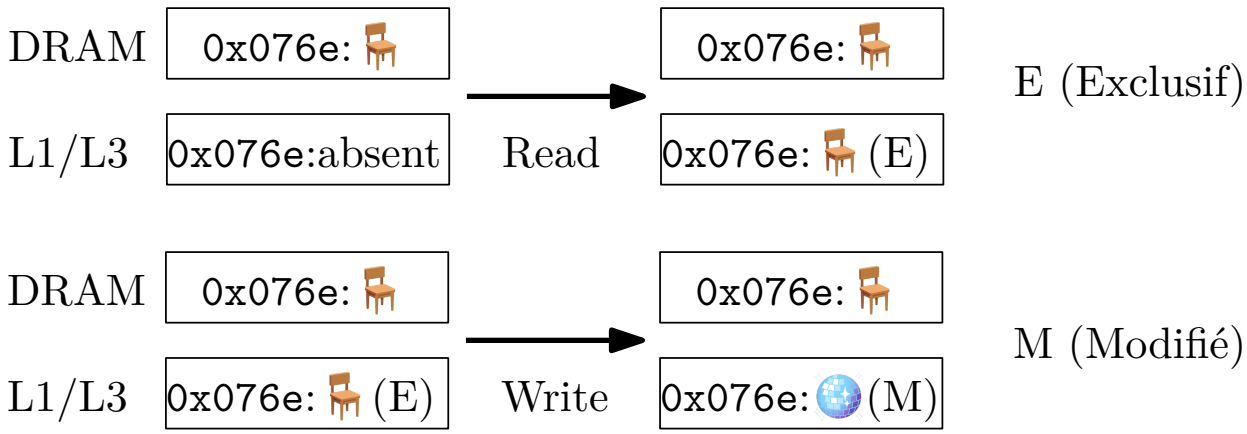
Pistes pour la suite

Merci de votre attention, questions ?

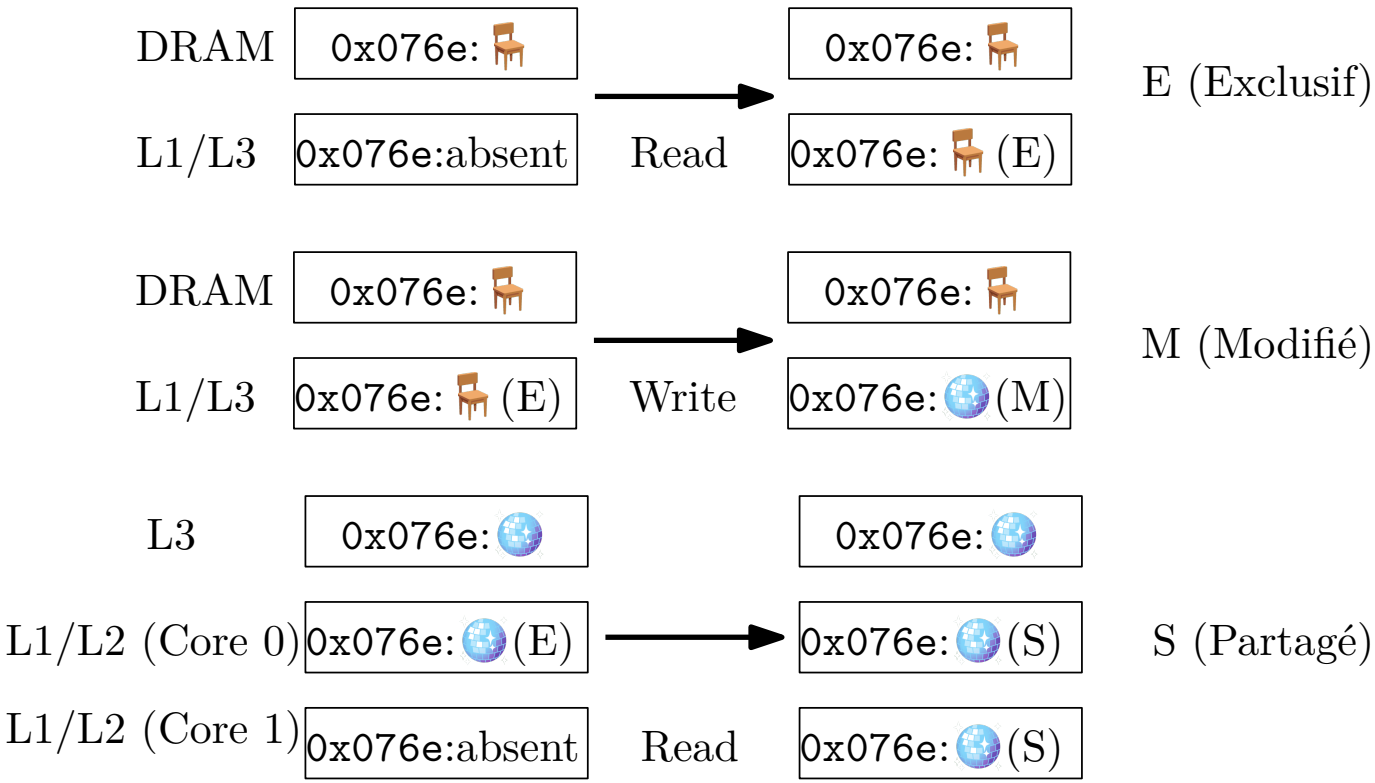
Cohérence de cache



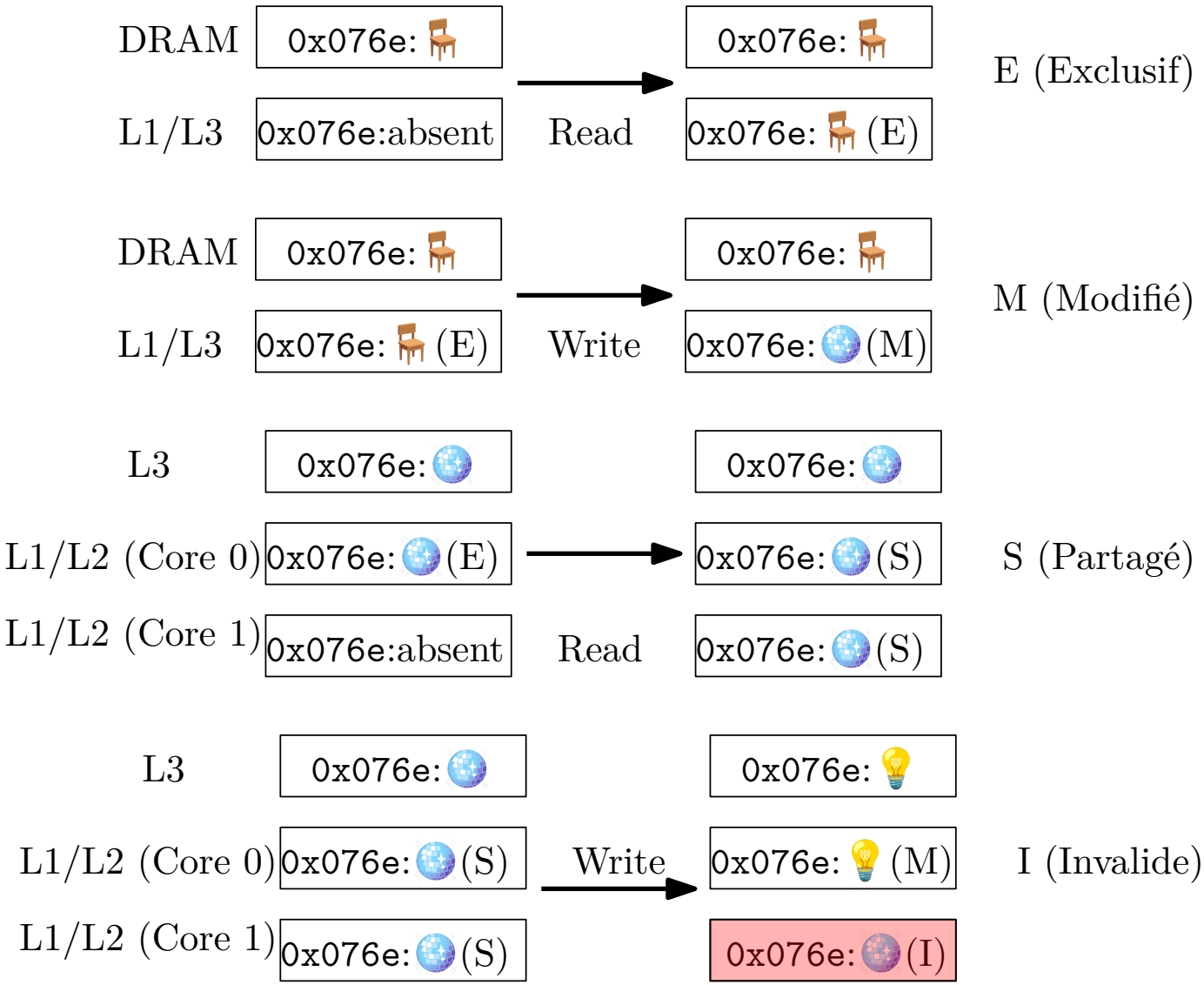
Cohérence de cache



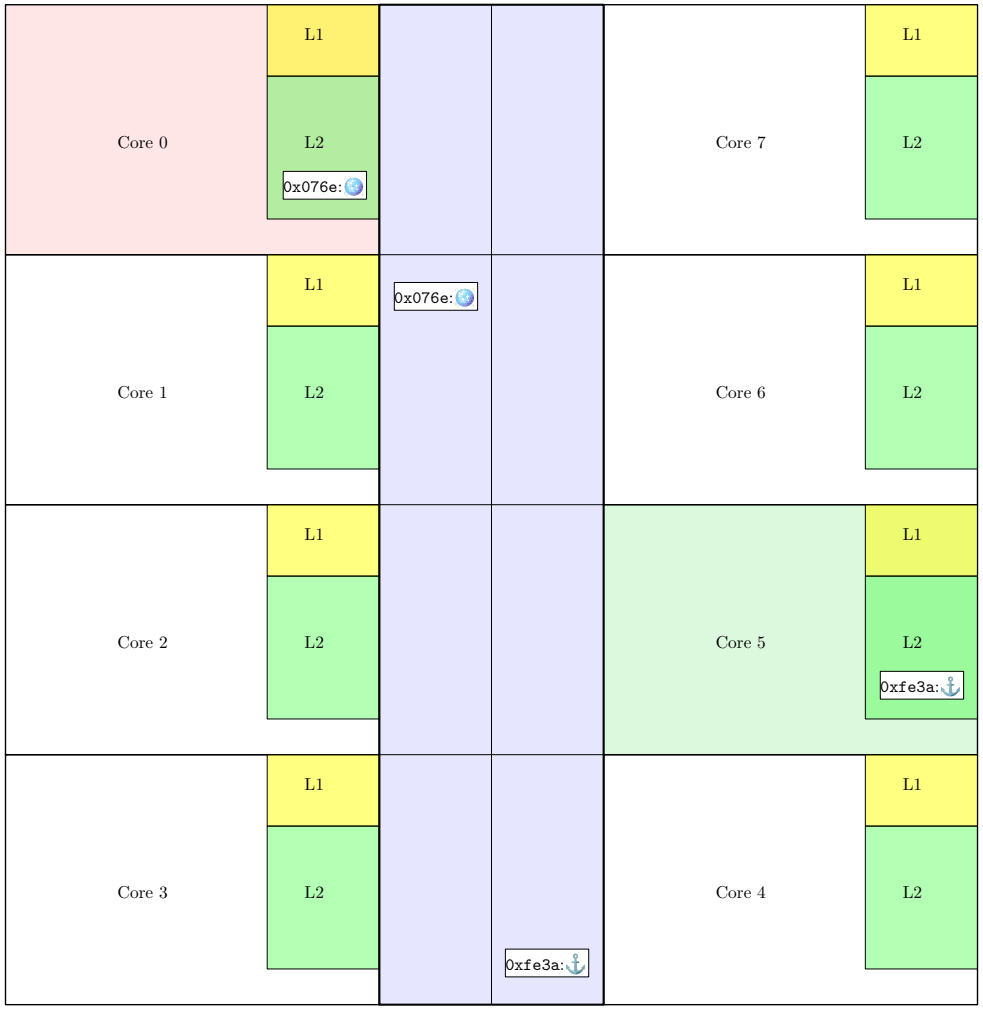
Cohérence de cache



Cohérence de cache

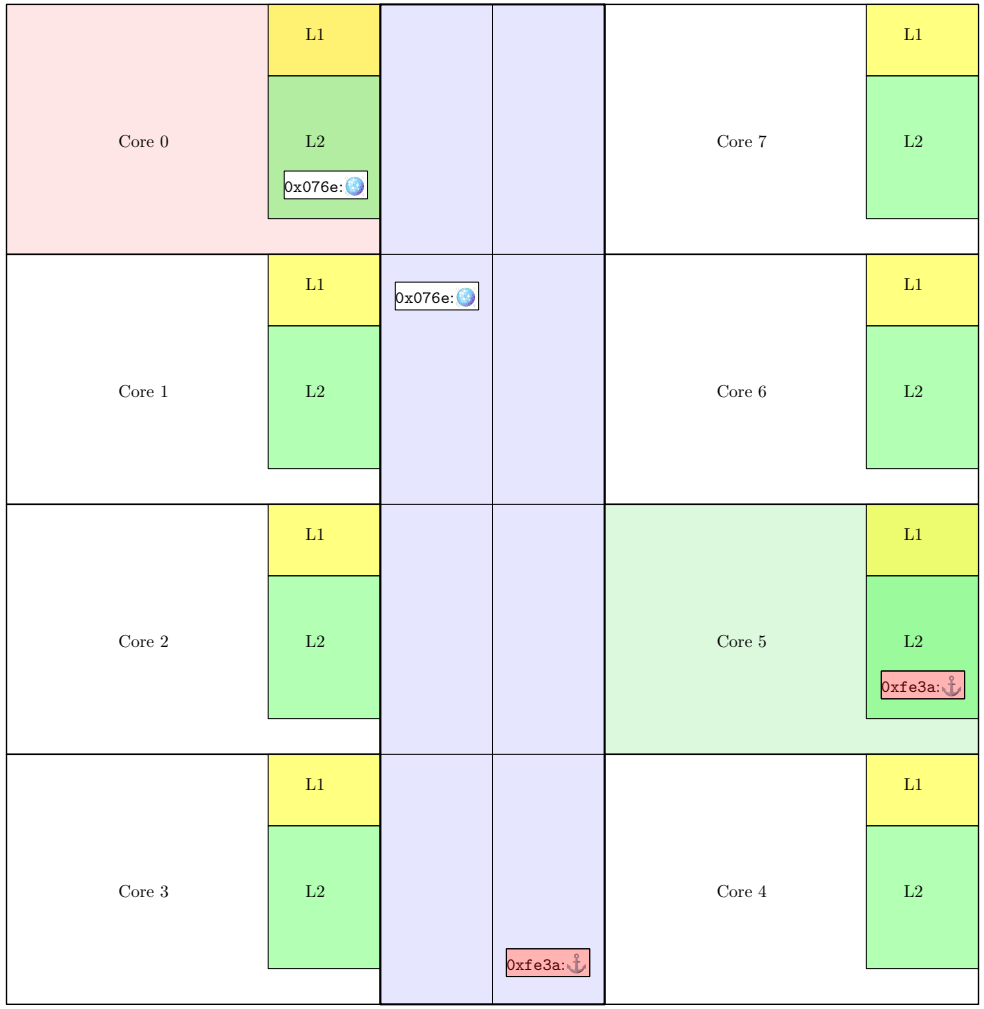


Flush+Reload



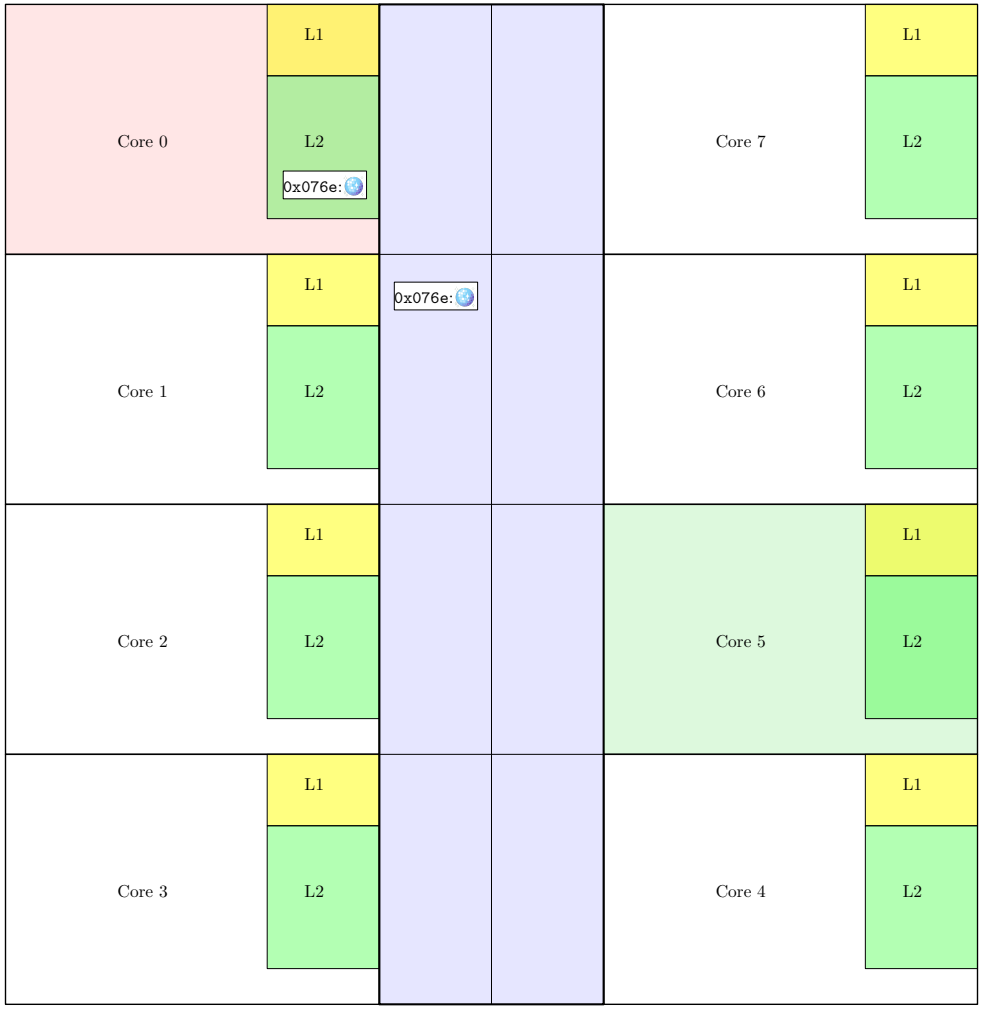
Observons le mécanisme proposé par Flush+Reload

Flush+Reload



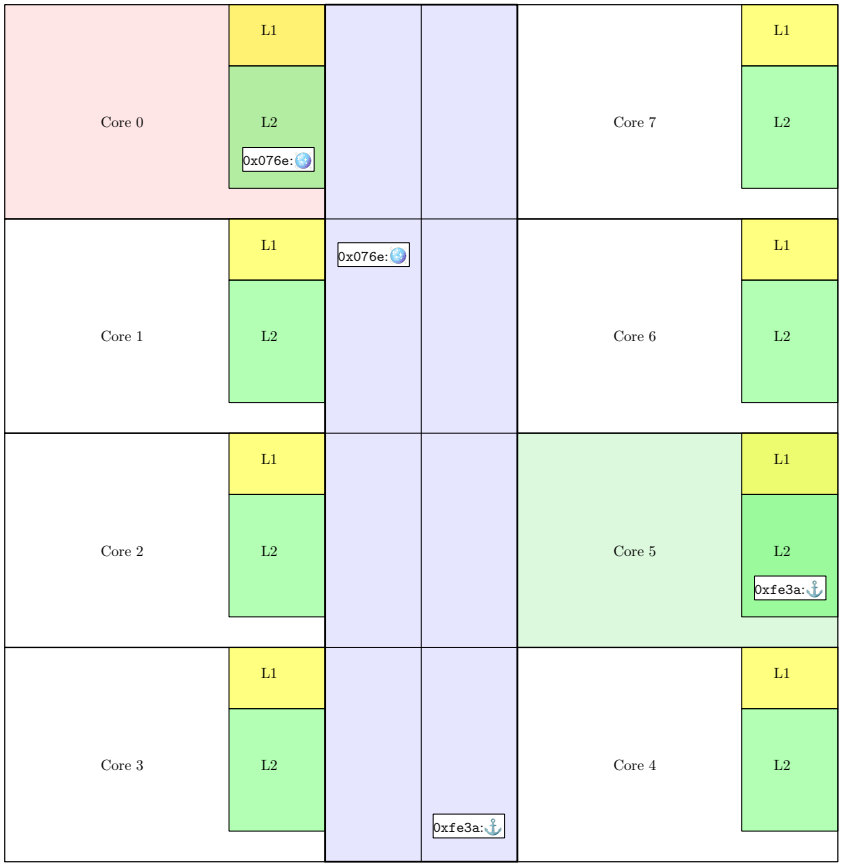
L'attaquant (en *Core 0*) appelle `clflush(0xfe3a)`

Flush+Reload

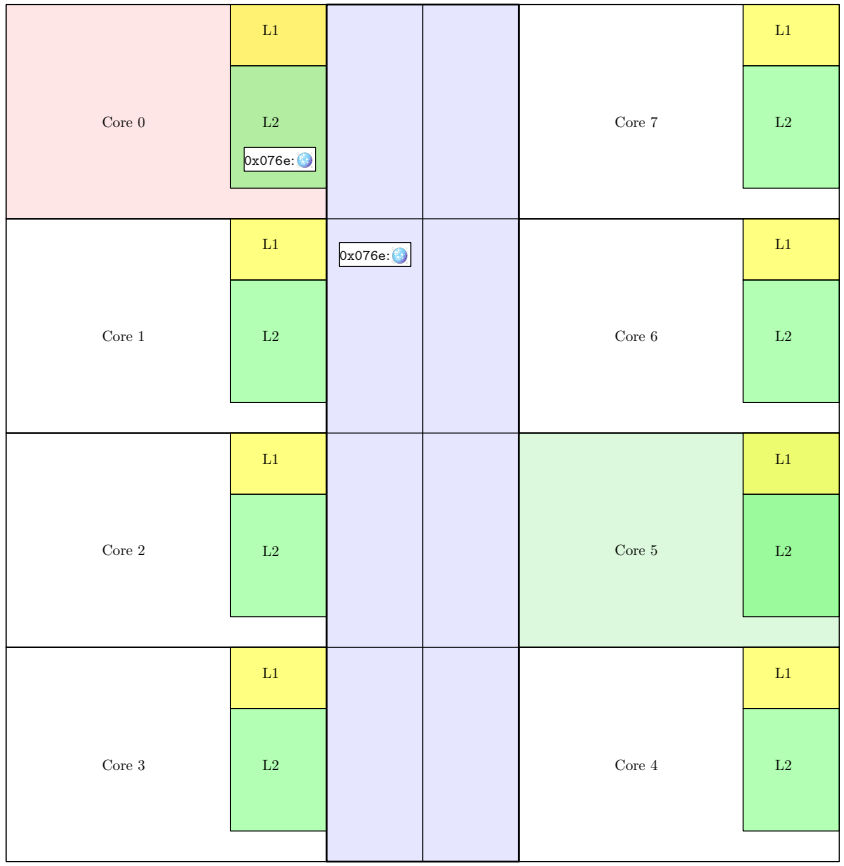


L'attaquant (en *Core 0*) appelle `clflush(0xfe3a)`

Flush+Reload



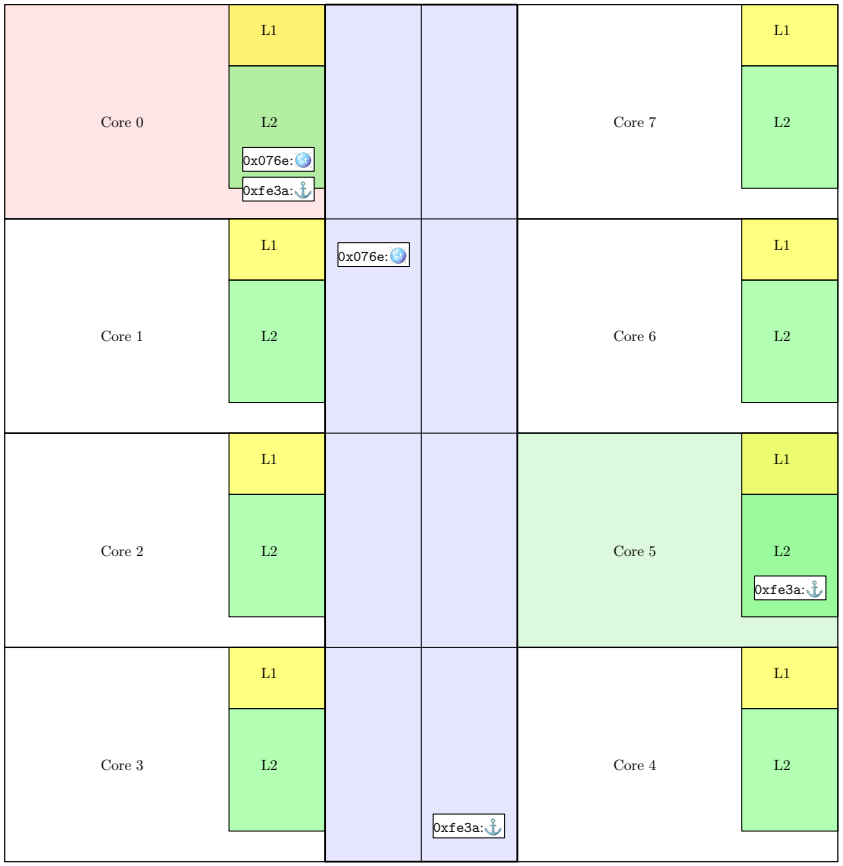
La victime charge `0xfe3a`



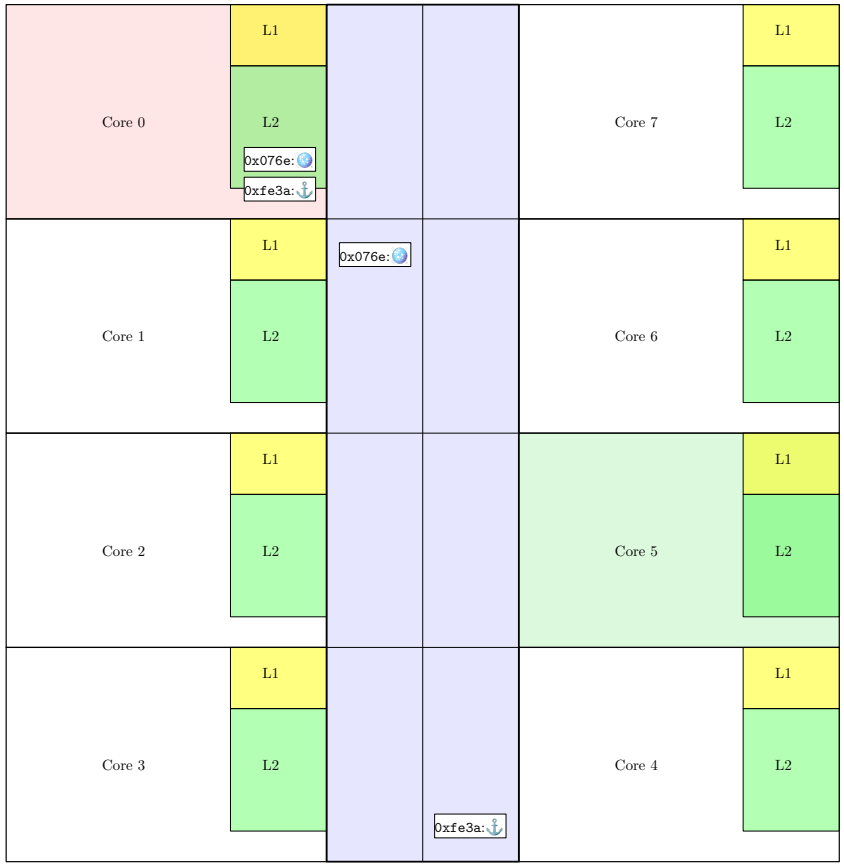
La victime ne fait rien

Flush+Reload

L'attaquant charge à nouveau `0xfe3a`, en mesurant le temps nécessaire



⏱️ ~1ns



⏱️ ~10-100ns

Flush+Reload sur un système multi-*socket*

Avec Flush+Reload, un *reload* en *hit remote* est potentiellement très long.
Est-ce que Flush+Reload fonctionnerait tout de même dans ce cas ?