Caractérisation de l'instruction `clflush` sur systèmes multi-*socket*

Augustin Lucas
ENS de Lyon

Encadré par :
Guillaume Didier, Angeliki Kritikakou
Équipe Taran
Laboratoire Irisa
Université de Rennes 1

3 juin 2024 - 12 juillet 2024

Résumé

Les attaques par canaux auxiliaires sont particulièrement intéressantes car elles permettent un accès à des informations restreintes, par delà même les *sandbox* ou machines virtuelles. Nous nous intéresserons plus particulièrement aux attaques par canaux auxiliaires sur le cache : Flush+Reload et Flush+Flush.

Ce stage cherche à renseigner la topologie des appels à l'instruction `clflush` sur des systèmes *Intel* à deux *sockets* ; notamment pour mieux évaluer l'état d'une ligne de cache en fonction du temps d'exécution de l'instruction `clflush` sur celle-ci.

Table des matières

1 Contexte

1.1 Hiérarchie de cache

La mémoire DRAM d'un ordinateur est lente comparée à la fréquence du CPU. Le CPU dispose donc de caches, basés sur des mémoires SRAM, plus petites mais plus rapides. Stocker en cache les éléments accédés le plus fréquemment permet donc de réduire le nombre d'appels à la mémoire et donc le temps d'exécution.

Les processeurs que nous étudions disposent de 3 niveaux de cache : L1, L2, L3. Chaque coeur possède, au premier niveau, un L1-I (cache des instructions) et un L1-D (cache des données). Au second niveau, il possède un L2 qui contient potentiellement des données et des instructions. Au dernier niveau, le L3 est partagé entre tous les coeurs de la *socket*, et celui-ci est inclusif de tous les caches de niveau inférieur : toute ligne stockée dans le cache L1 ou L2 est également dans le L3.

Si le L3 est partagé, il n'est cependant pas situé en un seul endroit dans le CPU mais est réparti en différentes *slices* : des tranches de mémoire accolées chacune à un coeur. Dans le modèle étudié, chaque coeur a exactement une *slice*.

Lorsqu'un coeur accède à une donnée qui n'est pas encore dans son cache, c'est toute la ligne de mémoire : les 64 octets environnants (généralement) qui sont chargés dans son L1 ou L2. Comme le L3 est inclusif, la ligne y est chargée également. Une fonction de hachage non-documentée attribue à chaque adresse physique une unique *slice* dans laquelle elle peut être mise en cache. Différents travaux [**?**, **?**] ont permis de déterminer cette fonction.

Si les fonctions initialement utilisées étaient simplement basés sur certains bits de l'adresse, la fonction de hachage utilisée à partir de la micro-architecture Sandy Bridge (et au moins jusqu'à Tiger Lake) utilise le XOR de plusieurs bits de chaque adresse physique pour générer chaque bit du numéro de *slice*. Alors, la fonction de hachage est linéaire sur les processeurs avec un nombre de coeurs qui est une puissance de 2, mais une composante non-linéaire s'ajoute sur les autres processeurs.

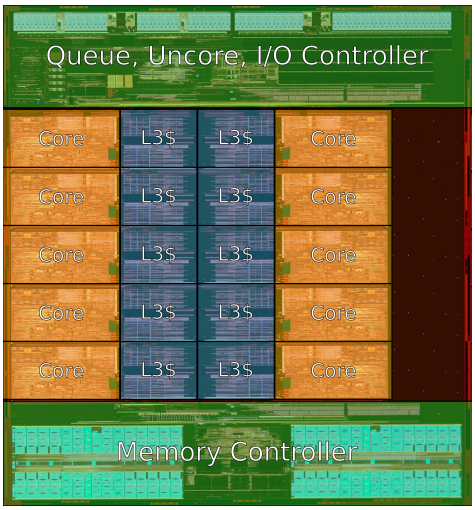

FIGURE 1 – Broadwell Deca-Core die shot by Intel - annotated by Wikichip [**?**]

1.1.1 *NUMA*

Dans un système à plusieurs *sockets*, chaque *socket* a de la mémoire DRAM à laquelle il peut accéder directement. Sur les machines Intel, le *QPI* (*QuickPath Interconnect*) permet à un processeur d'accéder aux données localisées dans la mémoire rattachée à un autre processeur.

Le principe *NUMA* (*Non-uniform memory access*) délimite alors des groupes contenant un processeur et les adresses mémoire qui en sont rapprochées et les transmet au système d'exploitation. Cela permet à ce dernier d'allouer la mémoire dans les adresses proche du processeurs qui en a besoin.

Sur Linux, si beaucoup d'accès mémoire sont réalisés d'un processeur à des adresses qui lui sont éloignées, les pages correspondantes sont migrées.

1.2 Protocoles de cohérence de cache

Dans des systèmes à plusieurs coeurs, d'autant plus avec plusieurs processeurs, où chaque coeur a un cache qui lui est propre, un problème de cohérence apparaît. Comment s'assurer qu'une ligne de données ne soit pas réécrite de différentes manières en plusieurs caches du système ?

On peut commencer par définir des états dans lesquels sont considérées les lignes de cache :

— *M* Modifié : la ligne est stockée modifiée dans un unique cache
— *E* Exclusif : la ligne est stockée intacte dans un unique cache
— *S* Partagé : la ligne est stockée intacte dans plusieurs caches (plusieurs caches disjoints, des L1 de coeurs différents par exemple)
— *I* Invalide : la ligne a été invalidée dans ce cache, car modifiée dans un autre cache par exemple

D'autres sont parfois ajoutés à cette liste comme *Forward*.

Une première solution au problème de cohérence de cache, dite par annuaire (*directory*), consiste à avoir à côté des différents caches un *directory* qui contient pour chaque ligne de mémoire en cache son état dans les différents endroits où elle est stockée. Lorsqu'une donnée partagée est modifiée, le *directory* est chargé d'envoyer aux autres caches une requête invalidant la ligne modifiée

Dans l'autre solution principalement utilisée, dite par *snooping*, chaque cache surveille de son côté les lignes qu'il a en mémoire.

Dans le système étudié, la cohérence se gère par *directory* au sein d'une même *socket*, mais par *snooping* entre les deux *sockets*.

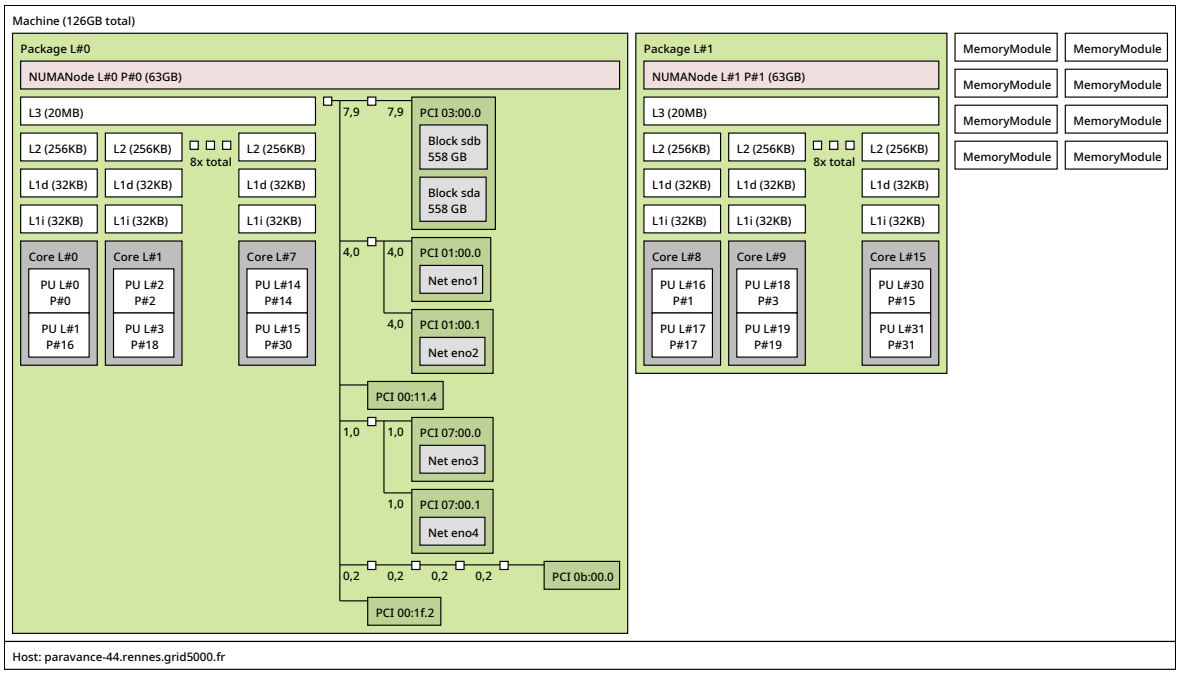

FIGURE 2 – Topologie *NUMA* de la machine *paravance* extraite avec l'outil `lstopo` de `hwloc`

1.3 Attaques par canaux auxiliaires

1.3.1 Mémoire partagée

Les systèmes d'exploitation utilisent le principe de mémoire partagée pour réduire l'utilisation totale de mémoire physique. C'est-à-dire que différentes pages de mémoire virtuelle correspondent à une même page de mémoire physique, partagée potentiellement entre plusieurs processus.

Par exemple, les bibliothèques utilisées par plusieurs programmes ne sont chargées qu'une seule fois en mémoire pour tous les programmes les utilisant.

De la même manière, lorsque un processus est dupliqué (via *fork*) ou lancé deux fois, les données qu'ils ont en commun (le code du programme par exemple) sont partagées entre les différentes instances du programme.

Une autre forme de déduplication consiste à regarder les pages de mémoire contenant les mêmes données et à les combiner. Cela peut amener différents processus *sandbox*és même de machines virtuelles différentes à partager des données en commun.

1.3.2 L'instruction `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.*

Lorsque l'instruction `clflush` est exécutée, l'adresse et la ligne de cache associée sont évincées de tous les caches L1, L2 et L3 où elles se trouvaient possiblement, et cela dans tous les *sockets* du système. Si des modifications avaient eu lieu, les modifications sont réécrites dans la mémoire DRAM.

L'instruction `clflush` est accessible à tout utilisateur non privilégié sur les adresses mémoires auxquelles il a accès.

1.3.3 Flush+Reload

Le temps de chargement d'une donnée est largement influencé par sa présence en cache. Mesurer le temps de chargement d'une adresse permet donc de déterminer aisément si la ligne de cache associée était déjà présente en cache.

Flush+Reload [?] propose donc la méthode suivante :

Data: x : addresse à surveiller
Result: Y a t-il eu un accès à x ?
$clflush(x)$; /* x n'est plus en cache */
$sleep(n)$; /* Si un coeur accède à x, x est à nouveau dans le L3 */
$t \leftarrow rdtsc()$;
$read(x)$;
$total\_time \leftarrow rdtsc() - t$;

Algorithm 1: Flush+Reload

La différence entre le temps de lecture depuis le cache et depuis la DRAM étant conséquent, cette méthode permet de déterminer avec un faible taux d'erreur si un accès a été fait à une adresse, le grand nombre de *reload* effectués par les `clflush` et `read` successifs est cependant visible via des compteurs de performance et donc détectable, et ne permet pas une haute fréquence d'observation.

1.3.4 Flush+Flush

Le temps d'exécution de l'instruction `clflush` dépendant de l'état de cohérence de la ligne de cache concernée, la connaissance de son temps d'exécution permet de la même manière de déterminer dans quel état était la ligne. Flush+Flush [?] propose la méthode suivante :

Data: x : addresse à surveiller
Result: Y a t-il eu un accès à x ?
$clflush(x)$; /* x est dans l'état I */
$sleep(n)$; /* x passe dans l'état E si un unique coeur y accède en lecture */
$t \leftarrow rdtsc()$;
$clflush(x)$;
$total\_time \leftarrow rdtsc() - t$;

Algorithm 2: Flush+Flush

Les avantages de cette méthode par rapport à Flush+Reload sont multiples :

— Aucun accès mémoire n'est réalisé pour surveiller l'adresse, ce qui rend les méthodes de détection qui comptent le nombre de *cache miss* inefficaces. [?] propose des solutions de détection alternatives mais montre qu'elles auraient toutes un coût bien trop élevé.
— Comme aucun accès mémoire n'est réalisé, la vitesse de traitement et le débit de données qui peuvent être extraites est bien plus élevé : 496KB/s contre 298KB/s pour Flush+Reload
— Comme l'opération mesurée agit sur tous les caches du système et pas seulement sur ceux utilisés par l'attaquant, Flush+Flush peut opérer dans un système avec une hiérarchie de cache non inclusive. Ce qui est le cas des systèmes à deux *sockets* notamment, alors que Flush+Reload nécessite de partager un cache en commun (le L3 par exemple).

Similairement à ces autres méthodes, Flush+Flush peut extraire des données du fonctionnement des autres processus en regardant les accès mémoires faits dans les bibliothèques partagées, qui occupent les mêmes zones de la mémoire physique pour différents processus.

Daniel Gruss et al. [?] proposent par exemple de récupérer le nonce d'une clé OpenSSL avec Flush+Reload en regardant les zones mémoire accédées pendant le chiffrement de données. Un en-

registreur de frappe (*keylogger*) basé sur les pages accédées dans la librairie GTK `libgdk.so` y est également mis en oeuvre.

Là où Flush+Reload choisit de mesurer le temps pour charger à nouveau une adresse en mémoire, Flush+Flush choisit de mesurer le temps nécessaire pour l'évincer : la différence entre un *cache hit* et un *cache miss* est alors beaucoup moins perceptible (moins de 12 cycles de CPU).

De bons résultats [**?**] ont toutefois été obtenus en appliquant un seuil global. Mais la connaissance des coeurs attaquant et victime et de la *slice* de la ligne surveillée permettent potentiellement une meilleure précision.

Dans un scénario réel, l'attaquant peut choisir le coeur sur lequel il s'exécute, a accès au coeur sur lequel le processus victime s'exécute via `/proc/pid`. La *slice* peut-être trouvée en utilisant l'adresse physique mais cette information demande généralement des privilèges qu'un attaquant n'aura pas. En revanche, si la fonction de hachage est linéaire – ce qui est le cas si le nombre de coeurs est une puissance de 2 (1.1) – il est possible de définir une classe d'équivalence des adresses des pages qui appartiennent aux mêmes *slices*. Une connaissance fine d'éléments dépendants du numéro de *slice* – comme le temps d'exécution de `clflush` dans certaines conditions – permettrait alors de déterminer la *slice*.

2 Motivation

Guillaume DIDIER et Clémentine MAURICE [**?**] proposent une rétro-ingénierie des messages échangés en fonction de l'état de cohérence des lignes évincées du cache. Cela passe par l'étude des sources de variabilité et permet de mieux choisir un seuil propre à chaque combinaison attaquant/victime/*slice*.

Ce travail s'était intéressé à certains processeurs Intel de micro-architectures *Coffee Lake* et *Haswell* à une seule *socket*, mais a révélé que les résultats seraient bien plus complexes sur des systèmes à plusieurs *sockets*.

3 Organisation des expériences

Les expériences ont été faites sur 6 machines différentes, via la plateforme Grid'5000 afin d'échantillonner au moins une machine par processeur répondant aux critères suivants :

— Processeur Intel ;
— Exactement 2 *socket* ;
— Nombre de coeurs par *socket* est une puissance de deux
— Micro-architecture antérieure à SkyLake [1]

Les machines suivantes ont donc été utilisées [**?**]

Nom	Nombre de coeurs	Processeur	Micro-architecture
rennes/roazhon11	16	Xeon E5-2660	Sandy Bridge
rennes/roazhon12	16	Xeon E5-2660	Sandy Bridge
rennes/parasilo	16	Xeon E5-2630 v3	Haswell
rennes/paravance	16	Xeon E5-2630 v3	Haswell
lyon/nova	16	Xeon E5-2620 v4	Broadwell
rennes/abacus2	16	Xeon E5-2609 v4	Broadwell

La prise de mesure consistait à relever pour un lot d'adresses (chacune correspondant à une *slice*) les temps d'exécution de l'instruction `clflush` pour toutes les paires de coeurs et tous les états de cohérence possibles.

Initialement, le turbo était désactivé sur toutes les machines, la fréquence des coeurs était fixée à la fréquence maximale en activant le mode "performance" via `cpufreq`.

1. Le L3 n'est plus inclusif sur les processeurs serveur à partir de SkyLake et la topologie devient plus complexe

Nous avons trouvé que :

— Contrairement aux machine *Intel Core*, le mode performance ne permet pas de fixer la fréquence de tous les coeurs à la fréquence maximale possible. Nous avons donc fixé la fréquence des coeurs à la fréquence minimale (généralement 400MHz) ;

— L'architecture *NUMA* impliquant de déplacer les pages exploitées principalement par un processeur distant, les adresses physiques changeaient durant l'expérience, ce qui rendait tous les résultats complètement incohérents. Nous avons donc désactivé le *NUMA balancing* via `numactl`, ce qui peut se faire pour le processus courant sans privilèges [2].

— Nous avons également fixé la fréquence de l'*uncore* via un MSR, mais cela n'a pas eu de grand impact sur la clarté des résultats.

Les fichiers de résultats bruts sont accessibles en ligne [**?**].

4 Analyse des résultats

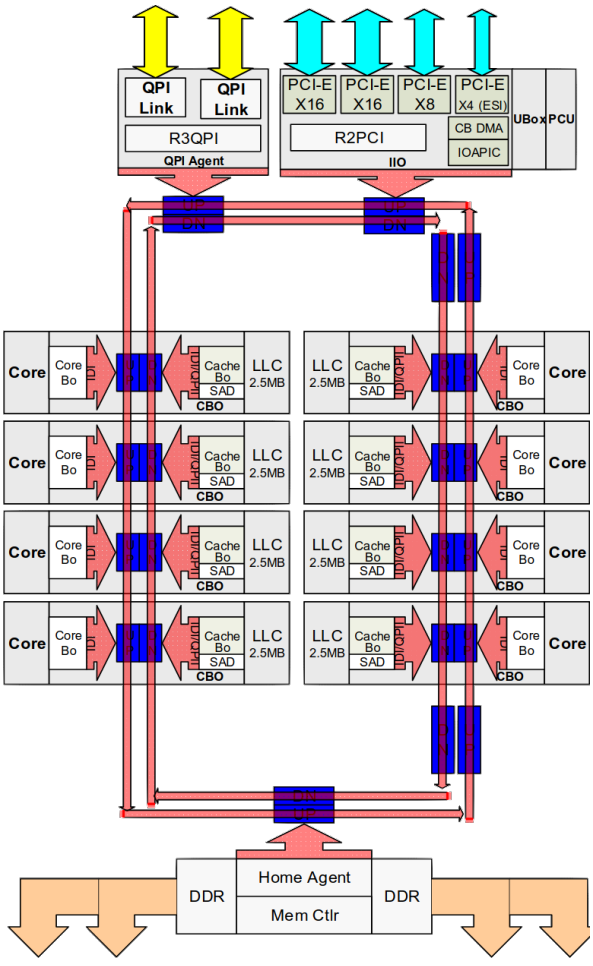

FIGURE 3 – Topologie LCC Haswell EP d'après [**?**]

Les schémas de présentation d'Intel suggèrent une topologie en anneau, avec un CPU divisé en deux grandes parties.

2. Il faudrait vérifier que cela fonctionne aussi pour les pages partagées comme les bibliothèques

4.1 Questions génériques

4.1.1 Numérotation des coeurs

Les premiers tests semblent révéler une numérotation suivant ce schéma [3] :

0	1
2	3
4	5
6	7

Pour simplifier les interprétations, nous les avons renumérotés de la façon suivante, c'est la numérotation que nous utiliserons pour la suite :

0	7
1	6
2	5
3	4

Pour trouver la numérotation des *slices*, deux méthodes sont possibles :
— Utiliser la méthode proposée pour un attaquant non-privilégié (1.3.4). Cela nécessite une connaissance préalable de la topologie ;
— Utiliser les compteurs de performance pour déterminer la *slice* d'une adresse. Cela nécessite de lire les MSR correspondants donc d'avoir un accès *root*. Nous avons suivi cette méthode afin d'être sûr que les numéros de *slice* correspondent aux numéros de coeurs.

4.1.2 Coeurs fantômes

Pour répondre à des demandes variées, *Intel* base ses processeurs serveurs sur trois tailles à compter de la micro-architecture Haswell :
— LCC (*Low Core Count*) : de 4 à 8 coeurs physiques pour la micro-architecture Haswell (jusqu'à 10 pour Broadwell)
— MCC (*Medium Core Count*) : de 10 à 12 coeurs pour Haswell (12 à 14 pour Broadwell)
— HCC (*High Core Count*) : de 14 à 18 coeurs pour Haswell (16 à 24 pour Broadwell)

Les machines que nous utiliserons sont uniquement basées sur la *die* LCC, mais certaines (les Broadwell à 8 coeurs par exemple) sont alors livrées avec des coeurs désactivés. La *slice* de cache L3 associée est alors désactivée également. Cela pose tout de même question sur la possibilité de sauts supplémentaires dûs à ces coeurs manquants : quel est le coeur désactivé ? Est-ce le même sur tous les processeurs du même modèle ? Quel est alors l'impact sur le temps d'exécution ? Par la suite, nous avons étudié principalement les résultats de la machine *paravance* pour éviter ces questions et établir plus facilement un premier modèle.

4.2 Topologie *miss*

Les résultats obtenus quand le *socket* attaquant et victime diffèrent suggèrent l'échange des messages suivants lors d'un `clflush` qui provoque un *cache miss* :

1. Le coeur attaquant contacte la *slice* locale suivant 5.
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. Il est possible que la numérotation soit en fait tournée à 180°. Comme nous avons appliqué la même rotation aux *slice*, cela n'affecterait que les positions respectives du QPI et du *Home Agent*

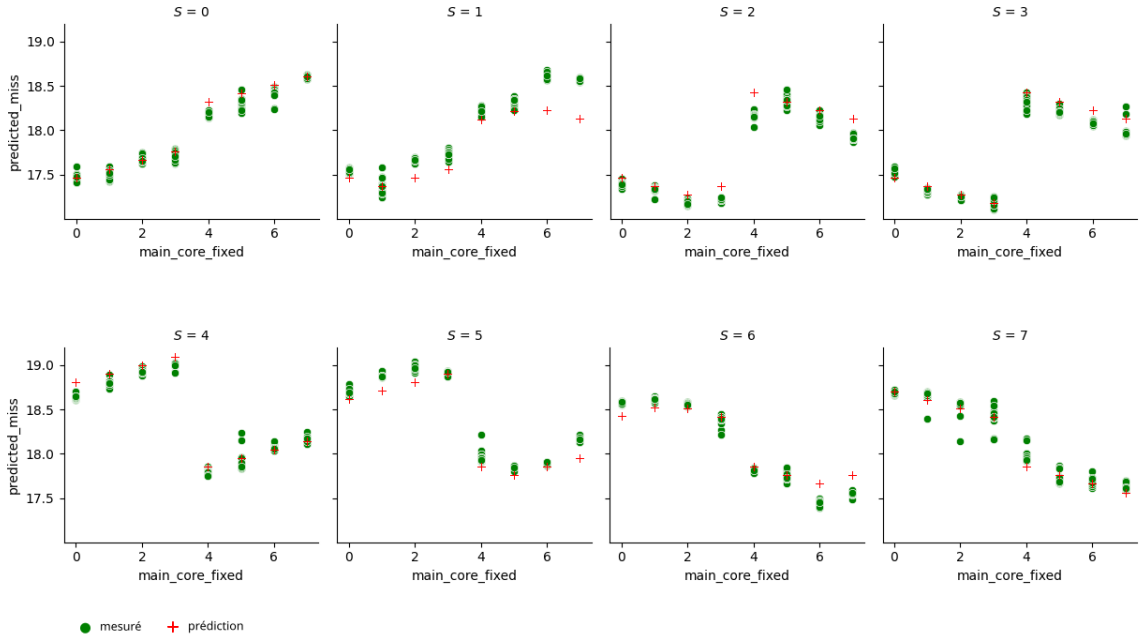

FIGURE 4 – durée prédite et réelle d'un cache *miss* selon les différents paramètres (*paravance*)

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. Nous n'avons effectivement pas trouvé d'élément permettant de montrer un trajet différent.
4. Le chemin est parcouru à l'envers pour repasser par la *slice* locale jusqu'au coeur attaquant

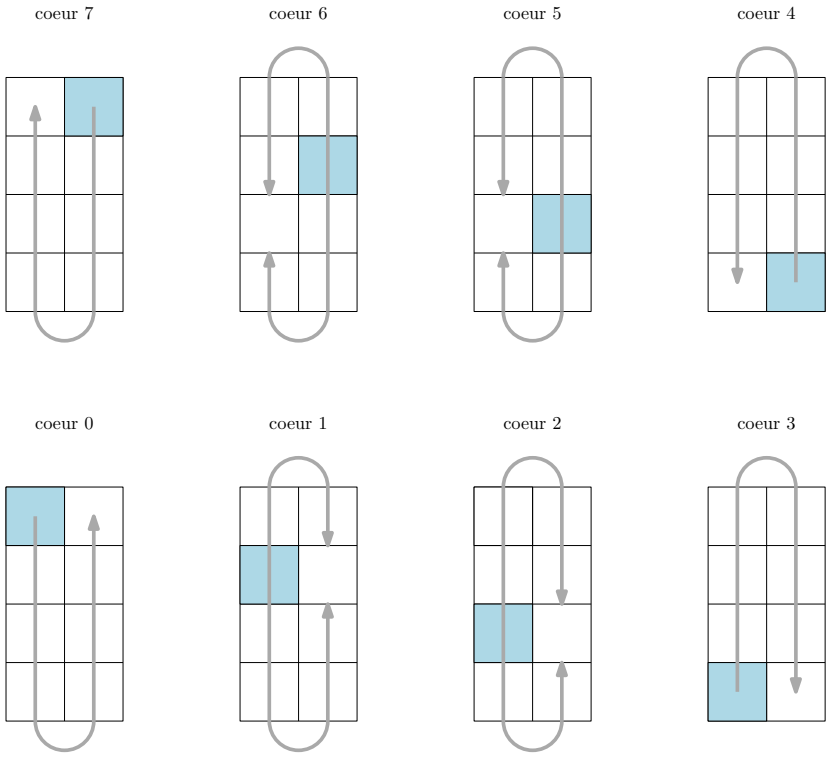

FIGURE 5 – trajet coeur attaquant → *slice* locale

Ainsi, le chemin privilégié pour le trajet coeur attaquant → *slice* locale serait assez proche du plus court, mais tenterait de limiter les passages par le *Home Agent* possiblement pour éviter de ralentir les opérations qui ont besoin de passer par lui.

4.3 Topologie *hit* exclusif

On peut diviser le *hit* en deux grands cas :
— L'attaquant et la victime sont dans la même *socket*
— L'attaquant et la victime ne sont pas dans la même *socket*

Initialement, le trajet de l'information est le même que dans le cas d'un *miss*, et ce n'est qu'au moment où le message parvient au niveau de la *slice* contenant la ligne en cache que les choses changent. La *slice* contacte alors le coeur victime, et le message fait le chemin retour une fois la ligne de cache évincée.

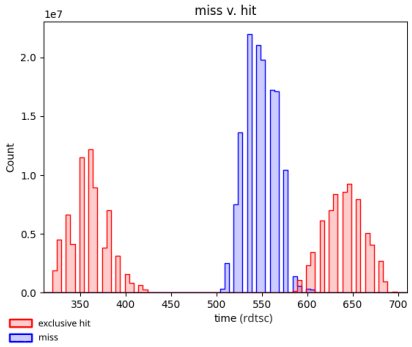

FIGURE 6 – Nombre de cache *hit* et cache *miss* selon la durée d'exécution de l'instruction `clflush`

Alors, on note une différence de coût significative entre un *hit* local et un *hit* distant, dû au passage par le *QPI*. Dans le cas où l'attaquant et la victime partagent la même *socket*, le coût d'un *hit* devient alors généralement beaucoup moins élevé que celui d'un *miss* ; contrairement au comportement sur des systèmes à une seule *socket*.

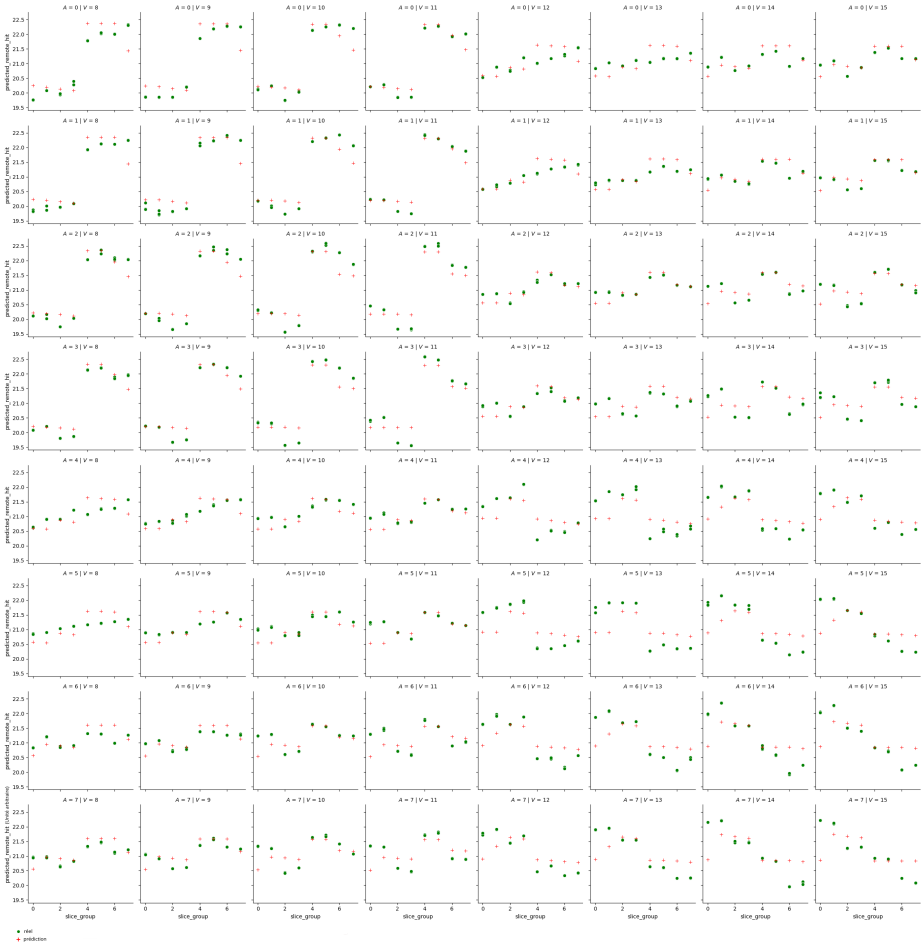

FIGURE 7 – durée prédite et réelle d'un cache *hit* selon les différents paramètres (*paravance*)

Si ce modèle est assez proche des données, on comprend que celui-ci est loin d'être parfait. La majeure difficulté dans l'établissement d'un modèle dans le cas d'un *cache hit* est la grande quantité de données : on ajoute une dépendance en une nouvelle variable, le coeur victime, qui n'était pas prise en compte pour un *miss*. De plus, les différents facteurs sont ici difficiles à décorréler, ce qui a rendu l'analyse plus chronophage.

5 Conclusion

Si nous n'avons pas pu trouver un modèle parfaitement cohérents avec les résultats analysés pour une question de temps, nous avons désormais une méthode pour prendre les mesures, tout en maîtrisant les principales sources de bruit.

Plusieurs pistes se proposent alors pour continuer le travail présenté ici :

— Affiner le modèle, qui n'est pas parfait sur les *cache miss* et ne colle pas bien aux *cache hit*. On pourrait ensuite voir comment celui-ci s'étend aux autres états de cohérrence de cache ;
— Clarifier les hypothèses avancées dans ce rapport en effectuant certaines vérifications : `numactl` permet-il bien de verrouiller les pages partagées ? Peut-on proposer une renumérotation automatique des slices ? Cela permettra de créer un modèle crédible d'attaquant ;
— Proposer une méthode de réassignation automatique des slices pour savoir depuis un utilisateur non privilégié la *slice* d'une adresse virtuelle ;
— Réaliser l'attaque Flush+Flush sur des systèmes à 2 *sockets*

Références

[1] Broadwell - microarchitectures - intel - wikichip (2024). https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_(client)#Deca-core_Broadwell.

[2] *SEC'14 : Proceedings of the 23rd USENIX conference on Security Symposium*, USA, 2014. USENIX Association.

[3] Intel Corporation. *Intel 64 and IA-32 Architecture Optimization Reference Manual*. 2024.

[4] Guillaume Didier and Clémentine Maurice. Calibration Done Right : Noiseless Flush+Flush Attacks. In *DIMVA 2021 - The 18th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Lisboa / Virtual, Portugal, July 2021.

[5] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush : A fast and stealthy cache attack, 2016.

[6] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks : Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[7] Augustin LUCAS Guillaume DIDIER. dendrobates-t-azureus, 2024. https://gitea.augustin64.fr/l3-ENSL/dendrobates-t-azureus/.

[8] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, 2013.

[9] Michael Klemm. Programming and tuning for intel xeon processors, 2015. https://docs.dkrz.de/_downloads/31658e1743d2d33f0664ae4c69d39360/Programming_and_Tuning_for_Intel_Xeon_Processors_2015-07-01_M.Klemm.pdf.

[10] Augustin LUCAS. g5k-nodes, 2024. https://gitea.augustin64.fr/l3-ENSL/g5k-nodes/.

[11] Augustin LUCAS. g5k-results, 2024. https://gitea.augustin64.fr/l3-ENSL/g5k-results/.

[12] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 48–65, Berlin, Heidelberg, 2015. Springer-Verlag.

Annexe

Contexte du stage

Ce stage a été effectué dans l'université de Rennes 1, dans le laboratoire de l'Irisa (Institut de Recherche en Informatique et Systèmes Aléatoires), l'un des plus grands laboratoires de recherche français dans le domaine de l'informatique avec plus de 850 personnes.

Il accueille 7 départements et une quarantaine d'équipes, j'ai été encadré par deux chercheurs de l'équipe Taran (Architectures matérielles spécialisées pour l'ère post loi-de-Moore) : Guillaume Didier et Angeliki Kritikakou. En discutant avec les autres membres de l'équipe, j'ai pu découvrir de loin des problématiques liées aux FPGA, l'ISA RISC-V et l'injection de fautes notamment.

J'ai beaucoup apprécié l'ambiance générale et tout ce que j'ai dû comprendre pour avancer sur ce sujet de recherche, mais j'ai regretté qu'il soit trop spécifique pour pouvoir demander des conseils facilement autour d'une pause café.

Contributions de code

Le code utilisé pour prendre les mesures a principalement été réutilisé de travaux existants [**?**]. J'y ai contribué en résolvant quelques problèmes liés aux machines utilisées, et en ajoutant un exécutable limitant la fréquence de l'*uncore*.

Le code que j'ai écrit ou auquel j'ai apporté des modifications consiste principalement en :

— Toute la logique d'analyse des résultats, afin d'afficher les données qui m'intéressaient et de comprendre les relations. Il est disponible dans [**?**] `cache_utils/*.py`

— Un environnement et un script pour déployer l'expérience automatiquement sur des noeuds de Grid'5000 [**?**]

Remerciements

Je remercie toute l'équipe Taran, Guillaume Didier et Angeliki Kritikakou tout particulièrement pour leur accueil chaleureux au sein du laboratoire de l'Irisa. Je suis très reconnaissant pour toutes les connaissances et l'expérience acquises durant ce stage.

Aknowledgements Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including Cnrs, Renater and several Universities as well as other organizations (see `https://www.grid5000.fr`).