PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure

# Étude de prefetchers matériels par canal auxiliaire sur le cache


Soutenue par
**Guillaume DIDIER**
Le 20 janvier 2023

Dirigée par
**David NACCACHE**



École doctorale n°386
**Sciences Mathématiques de Paris Centre**

Spécialité
**Informatique**

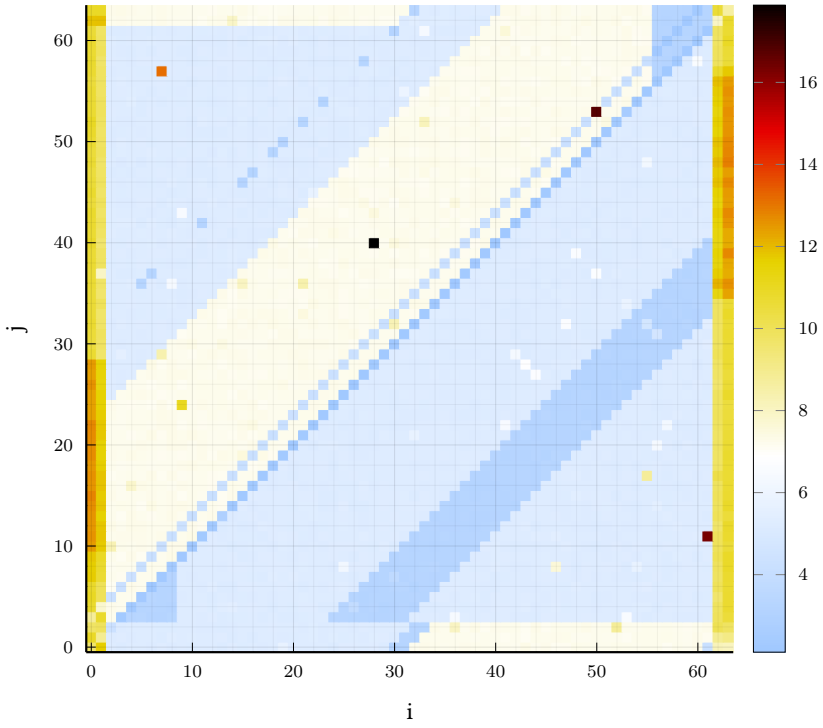

Composition du jury :

| | |
|---|---|
| David MONNIAUX<br>Directeur de Recherche, CNRS | *Président* |
| Guy GOGNIAT<br>Professeur, Université Bretagne Sud | *Rapporteur* |
| James HOE<br>Professeur, Carnegie Mellon University | *Examinateur* |
| Biswabandan PANDA<br>Professeur Assistant, IIT Bombay | *Examinateur* |
| Angeliki KRITIKAKOU<br>Maitresse de Conférence,<br>Université de Rennes 1 | *Examinatrice* |
| Clémentine MAURICE<br>Chargée de Recherche, CNRS | *Encadrante de thèse* |
| David HÉLY<br>Maitre de Conférence, Grenoble INP,<br>Univ. Grenoble Alpes | *Rapporteur, invité pour la soutenance* |
| David NACCACHE<br>Professeur, École Normale Supérieure | *Directeur de thèse, absent* |

ENS | PSL

Thèse de doctorat :
Étude de prefetchers matériels par canal auxiliaire sur le cache

Ph.D. Thesis:
*Studying hardware prefetchers using cache side channels*




Guillaume Didier

# Remerciements — Thanks

Je tiens tout d'abord à remercier Clémentine Maurice pour m'avoir accompagné et encadré, au quotidien, pendant ces trois ans d'aventure. Merci pour ces trois ans ‼! Je remercie aussi David Naccache pour avoir bien voulu endosser la responsabilité administrative de directeur de thèse[1].

Je remercie ensuite mes rapporteurs, David Hély et Guy Gogniat, pour avoir pris le temps de lire ce manuscrit, qui n'était pas des plus courts. Je remercie aussi David Monniaux pour avoir présider le jury et géré l'empêchement de dernière minute de mon directeur, et David Hély qui a accepté de devenir membre invité pour compenser l'absence de David Naccache. *I also thank all of the jury members, in particular James Hoe and Biswabanda Panda who attended remotely despite the time differences. I also thank James for allowing me to run a few preliminary experiments on one of his machines while at CMU.*

Je remercie aussi la DGA, qui m'a permis de faire cette thèse, en particulier David Lubicz[2], l'ENS Ulm pour avoir été mon école doctorale et l'IRISA, pour m'avoir accueilli en son sein, à Rennes, pendant trois ans, et en particulier les équipes successives EMSEC et SPICY.

Je remercie ensuite l'ensemble de mes collègues de l'équipe EMSEC, puis des équipes CAPSULE et SPICY, avec en premier lieu mes co-bureaux, Thomas, Daniel, Agathe (brièvement), et tous les autres doctorants, Victor, Angèle, Joshua, Alexandre, Diane, Gwendal, Pierrick, Clément, entre autres, Post-Doc et Permanents. Merci plus particulièrement à Pierre-Alain, Gildas et Stéphanie pour m'avoir accueillis dans leurs équipes.

Un très très grand merci aussi à Aurélie, notre formidable assistante d'équipe. Comment ferions-nous sans toi ?

Je remercie évidemment mes parents et ma famille, en particulier ma mère pour certaines recettes de cuisines, et pour la première soutenance à laquelle j'ai assisté et la première thèse que j'ai feuilletée.

Je remercie enfin mes amis, à Rennes, notamment à l'aumônerie de Beaulieu ainsi qu'à Vocal'Insa et au FOG qui m'ont accueilli ; comme ailleurs, Faerix, *friends from overseas*, ArtifiX et tous ceux avec qui j'ai gardé le contact.

[1] Absent le jour de la soutenance suite à un grave problème familial de dernière minute

[2] Attention, un David peut en cacher un autre.

# Contents

# Résumé

## 1. Contexte

### 1.1. Pourquoi effectuer la rétro-ingénierie de processeurs d'ordinateurs ?

Les ordinateurs traitent des informations et communiquent avec le monde extérieur en suivant des programmes stockés en mémoire. On appelle *entrées/sorties* les communications avec l'extérieur. Au cœur de ces appareils, les *processeurs*, aussi connus sous l'acronyme anglais *CPU* (*Central Processing Units*), exécutent les instructions de traitement. Les processeurs modernes que nous étudions sont des circuits intégrés à très grande échelle, comportant aujourd'hui des milliards voire des dizaines de milliards de composants électronique de taille manométrique appelés *transistors* sur un rectangle de quelques centimètres carrés de silicium. Les processeurs sont souvent décrits comme le cerveau des ordinateurs, quand bien même les ordinateurs ne sont certainement pas doués d'intelligence. L'aphorisme « Les ordinateurs font très rapidement et précisément des erreurs. », issue de Usenet, résume probablement bien le fait que les ordinateurs exécutent à la lettre les instructions, souvent erronées, que leur donnent les humains, mais le font de façon très précise et rapide.

Même si les principes généraux de conception de ces circuits sont assez connus, les entreprises qui conçoivent des processeurs à la pointe de la technologie, comme Intel, AMD, IBM, ARM ou Apple, sont assez réticentes à révéler le fonctionnement de leurs processeurs et les techniques employées pour augmenter leurs performances. Au lieu de cela, ils documentent uniquement comment les utiliser et gardent secret ce qui se cache sous le capot. Pourtant, il existe de nombreuses raisons de regarder sous le capot.

Tout d'abord, on ne peut pas compter sur la compétence des fabricants pour assurer la sécurité. Ces processeurs sont au cœur de nos ordinateurs et manipulent nos données, par définition, privées et précieuses. Il est donc impératif d'évaluer la sécurité de ces systèmes sans se reposer sur leur fabricant. Pour cela, il faut, de manière indépendante, à la fois comprendre leur fonctionnement et vérifier leur comportement. En raison de la complexité de ces systèmes, il a été démontré que des erreurs sont régulièrement commises, introduisant involontairement des problèmes de sécurité potentiels [18, 45, 53, 62, 65, 129].

Par ailleurs, une compréhension précise des processeurs est aussi précieuse pour qui étudie les performances des logiciels et du matériel informatique qu'elle ne l'est pour le chercheur en sécurité. La capacité à modéliser avec précision le comportement des différents sous-systèmes d'un processeur peut être essentielle pour les chercheurs qui proposent de nouvelles techniques tandis que la compréhension du matériel est nécessaire pour tenter de repérer un goulot d'étranglement dans une charge de travail [13].

Enfin, la rétro-ingénierie est aussi nécessaire lorsque le fabricant ne documente même pas l'interface de ses produits, appelée l'*architecture*. C'est, par exemple, le cas des processeurs

conçus par Apple connus sous le nom d'Apple Silicon. L'équipe qui écrit des pilotes pour permettre à Linux de fonctionner sur ces systèmes doit commencer par faire la rétro-ingénierie des interfaces matérielles. Dans notre cas, cependant, nous nous intéressons à des processeurs fabriqués par Intel, dont l'architecture est documentée [47]. Cependant, l'implémentation de cette interface, la *micro-architecture*, est en grande partie non documentée.

Il est donc utile d'étudier les processeurs et de découvrir les détails non divulgués par les fabricants. La *rétro-ingénierie* est le nom donné à ces tentatives visant à déterminer comment un dispositif ou un programme fonctionne sans aucun accès à sa structure interne ou à sa documentation interne.

## 1.2. Que sont les préchargeurs mémoire ou *prefetchers* ?

Les processeurs modernes sont assez complexes ; la micro-architecture comprend des sous-systèmes divers et nombreux. Parmi les composants qui rendent les processeurs modernes rapides, on trouve les mémoires caches, des éléments de mémoire petits et rapides qui accélèrent la plupart des accès à la mémoire principale, plus grande et plus lente. Les caches font partie du sous-système mémoire, qui a un impact significatif sur les performances. Par conséquent, de nombreuses techniques ont été développées pour améliorer les performances dudit système mémoire. L'une d'entre elles est le préchargement mémoire par le matériel. Elle s'appuie sur une classe de composants appelés *préchargeur* matériels, en anglais *prefetcher*, qui tentent d'anticiper les requêtes mémoires afin de réduire le temps nécessaire pour les satisfaire. Appartenant à la partie immergée de l'iceberg qu'est la micro-architecture d'un processeur, l'existence de ces préchargeurs est à peine divulguée et ils sont, pour la plupart, non documentés.

Pour illustrer l'action d'un préchargeurs à l'aide d'un exemple à échelle humaine, prenons un cuisinier typiquement français, occupé à préparer, par exemple, une blanquette de veau ou une tarte à la myrtille.[1] Il lui faut aller chercher les ingrédients, par exemple, dans le garde-manger, le réfrigérateur ou le potager. Tous ces allers-retours pour aller chercher les ingrédients prennent du temps. Imaginons maintenant que notre cuisinier ait un assistant. Il peut lui demander d'apporter tous les ingrédients et de les disposer sur la table, plus proche de lui. La table correspond, dans la métaphore culinaire, aux mémoire caches des ordinateurs. Un assistant intelligent peut deviner la recette que le cuisinier est en train de préparer et rassembler les ingrédients avant qu'il n'en fasse la demande. Il s'agirait là d'une illustration du préchargement mémoire, par ailleurs plus savoureuse que les données traitées par les ordinateurs.

## 1.3. Des fuites d'informations à travers la micro-architecture

L'étude de ces préchargeurs est entravée par un obstacle majeur : Ceux-ci opèrent sur les caches du processeur, qui font partie de la micro-architecture. Par conséquent, il est impossible d'interroger directement l'état des caches et d'observer leur fonctionnement. Cependant, l'état du cache modifie le temps d'exécution, et il est donc possible de déduire l'état du cache à partir des mesures de temps d'exécution. Cette façon d'obtenir des informations cachées à partir du temps d'exécution d'un programme est un exemple d'une classe de vulnérabilités plus large appelée « canaux auxiliaires » (*side channel* en Anglais), lorsque certaines informations privées

[1] *c.f.* Annexe B

fuient à travers les conséquences physiques (e.g., bruit, émissions électromagnétiques, temps d'exécution) de l'implémentation d'un algorithme, par exemple.

Pour prendre un exemple à l'échelle humaine, supposons que mon voisin chauffe sa maison avec une cheminée et que nous sommes en plein hiver. Je ne peux pas voir par-dessus la clôture qui sépare les deux maisons, et je ne peux donc pas savoir si mon voisin est chez lui ou en vacances. Cependant, en hiver, je peux observer ce qui sort de sa cheminée. S'il en sort de la fumée, mon voisin est probablement à la maison ; s'il n'y a pas eu la moindre fumée de la journée, mon voisin est probablement absent. À première vue, ce canal auxiliaire ne semble pas très nuisible, alimentant au pire les commérages de voisinage, à moins que vous ne prépariez un cambriolage. Cependant, les canaux secondaires peuvent souvent être utilisés à des fins plus néfastes. Un exemple bien connu dont la plupart des lecteurs auront probablement entendu parler est l'obtention de la combinaison d'un coffre-fort en écoutant les bruits émis en tournant ses boutons. En matière de sécurité informatique, de nombreuses sources de fuites peuvent exister, notamment le temps d'exécution, la consommation d'énergie, les émissions électromagnétiques ou le son émis. Une attaque par canal auxiliaire se produit lorsqu'un attaquant exploite un canal auxiliaire pour violer les garanties de sécurité et avoir accès à des informations qu'il ne devrait pas avoir.

Cette étude a été initiée dans un contexte où la sécurité des micro-architectures était réexaminée à l'aune d'une série d'attaques très médiatisées [17, 18, 35, 54, 62], dont la plupart appartiennent à la nouvelle classe des *attaques par exécution transiente*. Cependant, le domaine de la sécurité des micro-architectures est beaucoup plus ancien, avec des travaux précurseurs majeurs à la fin des années 90 et au début des années 2000 [11, 55, 90], présentant les premières attaques temporelles et attaques sur le cache, ciblant généralement des algorithmes de cryptographie. Les attaques par canal auxiliaire de cache sont des attaques micro-architecturales qui fournissent des informations sur l'état du cache. Ces attaques ne révèlent pas les valeurs dans le cache, mais divulguent l'identité des lignes qui sont mises en cache. Pour déduire l'action des *prefetchers*, nous avons besoin d'informations sur l'état des caches du processeur ; nous utiliserons donc de tels canaux auxiliaires sur le cache, à savoir les primitives Flush+Reload [135] et Flush+Flush [36], qui s'appliquent aux processeurs que nous étudions.

## 1.4. Des canaux sur le cache : Flush+Reload et Flush+Flush

Le domaine des canaux micro-architecturaux sur le cache a connu un développement important dans les années 2010, grâce, notamment, à la publication de l'attaque Flush+Reload en 2014 [135]. Elle s'appuie sur l'instruction `clflush`, qui demande au processeur d'assurer que la mémoire contient la dernière valeur d'une ligne de cache et de retirer cette ligne de l'ensemble des caches.

Les caches manipulent la mémoire à la granularité d'une *ligne* de cache, un groupe d'octets de taille fixe. L'instruction `clflush` ayant été introduite par Intel en 2000, l'attaque Flush+Reload publiée en 2014 a touché un large éventail de processeurs. Dans l'attaque Flush+Reload, une ligne de cache donnée est d'abord évincée des caches vers la mémoire principale puis rechargée. Le temps nécessaire pour la recharger permet de savoir si un autre processus a ramené cette ligne dans le cache entre-temps, les accès à partir de la mémoire principale étant beaucoup plus lents que les accès à partir du cache. Ce canal auxiliaire nécessite que les deux processus partagent une partie de leur espace mémoire, généralement en lecture seule ; une hypothèse

souvent vérifiée pour le code et les données en lecture seule. Après la mesure, la ligne de cache peut être évincée et rechargée à nouveau, répétant le processus. Cette primitive a été un outil fondamental pour l'essor de la recherche sur la sécurité des micro-architecture et est utilisée par la plupart des attaques fortement médiatisées appartenant au domaine de l'exécution transiente.

Sur les processeurs d'Intel, l'instruction `clflush` prend un temps différent pour s'exécuter selon qu'une ligne est présente dans le cache ou non, ce qui permet de construire la primitive Flush+Flush [36], découverte en 2016. Par rapport à Flush+Reload, l'instruction de lecture mémoire dont on mesure le temps d'exécution et l'instruction `clflush` qui suit sont toutes deux remplacées dans cette attaque par une seule instruction `clflush` dont on mesure le temps d'exécution. Sous la même hypothèse que Flush+Reload, cette primitive est plus rapide, la lecture depuis la mémoire ayant été supprimée, mais plus bruyante, la différence de temps d'exécution de `clflush` entre les lignes mises en cache et les lignes non mises en cache étant beaucoup plus petite. Malheureusement, le bruit a augmenté avec l'augmentation du nombre de cœurs des processeurs Intel depuis 2016.

Notre métaphore culinaire peut illustrer ces deux variantes : nous avons maintenant plusieurs cuisiniers, chacun avec son propre espace de travail, et des assistants qui vont et viennent entre les cuisiniers, la table principale de la cuisine (le cache), le garde-manger et le réfrigérateur (la mémoire principale). La table principale de la cuisine est partagée, mais les cuisiniers ne savent pas précisément ce qui s'y trouve ; seuls les assistants voient la table. Lorsqu'un assistant va chercher un ingrédient, cela prendra plus ou moins de temps, selon que l'ingrédient est déjà sur la table ou non. Ce temps de récupération est l'équivalent, dans la métaphore, du temps de lecture dans Flush+Reload. Si nous supposons qu'un cuisinier peut également demander à un assistant de remettre quelque chose à sa place, au motif que « Je n'en ai plus besoin », l'équivalent d'un `clflush`, tous les ingrédients d'une attaque Flush+Reload sont présents. Ainsi, un cuisinier peut déterminer ce que les autres cuisiniers préparent en observant le temps que prend l'assistant pour ramener chaque ingrédient après avoir ordonné que ces ingrédients soient retirés de la table. Pour la primitive Flush+Flush, il est logique qu'un assistant prenne un temps différent pour remettre à sa place un ingrédient qui est sur la table par rapport à un ingrédient qui est encore rangé à sa place. Le cuisinier curieux peut donc demander de façon répétée à son assistant de ranger les ingrédients et en déduire ceux qui sont utilisés.

## 1.5. État de l'art de la rétro-ingénierie des préchargeurs

La sécurité des micro-architectures nécessite une compréhension fine des processeurs et est donc allée de pair avec les progrès dans le domaine de la rétro-ingénierie des micro-architectures. Beaucoup de ces travaux se sont concentrés sur les caches [34, 39, 48, 59, 60, 67, 85, 92, 123, 134, 137], une cible de choix pour les attaques par canal auxiliaire avec un état persistant. Cependant, d'autres composants ont également fait l'objet d'une rétro-ingénierie et sont devenus des cibles d'attaque. Le problème physique *Rowhammer* [53], où des inversions de bits peuvent être déclenchées dans des emplacements adjacents de la mémoire DRAM, a conduit à des efforts de rétro-ingénierie de la DRAM [91]. De même, les prédicteurs de branchement ont fait l'objet d'une rétro-ingénierie [12, 31, 118], et la communauté des chercheurs a également examiné les structures utilisées pour planifier l'exécution des instructions. Tous ces efforts ont généralement abouti à des attaques micro-architecturales.

L'industrie a divulgué peu d'informations relatives aux préchargeurs. Intel ne fournit que les noms et une brève présentation de ses quatre préchargeurs dans son manuel d'optimisation [46]. Alors que quelques études académiques [19, 39, 105] ont étudié le *L1 stride prefetcher*, préchargeur de niveau 1 par pas, une seule étude [94] a examiné le *L2 Stream prefectcher*, préchargeur de niveau 2 pour les flux. Ils ont utilisé Flush+Reload pour obtenir un premier ensemble de résultats de rétro-ingénierie suffisants pour construire une attaque, même s'ils n'ont pas modélisé entièrement ce *prefetcher*. Un défi fondamental dans toutes ces études est qu'elles utilisent généralement les instructions de lecture comme primitive de mesure, ce qui peut influencer le préchargeur et limite la quantité de données qui peuvent être extraites d'une seule expérience.

### 1.6. Comment utilisons-nous les canaux auxiliaires sur le cache pour la rétro-ingénierie de préchargeurs ?

Maintenant que nous avons introduit ces canaux auxiliaires, comment peuvent-ils nous aider à faire la rétro-ingénierie des préchargeurs ? Dans notre métaphore, le préchargeur serait un assistant intelligent qui, en réponse à la série d'ingrédients demandés par le cuisinier, apporte quelques ingrédients supplémentaires sur la table, parce qu'il lui semble probable que le cuisinier en aura bientôt besoin. Ainsi, dans notre cuisine, le cuisinier pourrait essayer de tester la façon dont son assistant effectue ces prédictions en demandant des ingrédients et en examinant ensuite ce qui se trouve sur la table. Sauf qu'il ne peut pas regarder directement la table. S'il utilise la technique Flush+Reload, en mesurant le temps que met l'assistant pour aller chercher certains ingrédients, il fera plus de demandes auxquelles l'assistant donnera suite, ce qui aura pour effet de précharger davantage d'ingrédients. De même, le fait de chronométrer les accès à la mémoire pour déterminer ce qui se trouve dans le cache influencera le préchargeur et pourra modifier les résultats de l'expérience. Flush+Reload est donc difficile à exploiter pour faire la rétro-ingénierie des préchargeurs. Cependant, comme `clflush` est une instruction peu commune, nous supposons que les préchargeurs pourraient ne pas y réagir, et donc être un outil plus approprié pour comprendre leur fonctionnement. C'est donc l'objectif principal de notre recherche : utiliser Flush+Flush pour identifier de manière fiable comment les préchargeurs (ou *prefetchers*) réagissent aux demandes. Dans la métaphore de la cuisine, l'idée est de demander quelques ingrédients, puis de dire à l'assistant de ranger chaque ingrédient, en prenant note de ceux qui prennent le plus de temps à être ranger, afin de déterminer quels ingrédients supplémentaires ont été préchargés.

Pour spécifier plus précisément le champ de notre investigation, nous étudions la micro-architecture des processeurs Intel à haute performance, sortis vers 2019, dérivés de la micro-architecture Skylake. Nous concentrons nos études sur le système de cache, avec les préchargeurs comme cible principale.

## 2. Objectifs

L'objectif général qui a orienté nos investigations peut être résumé comme suit :

**Comment pouvons-nous utiliser les canaux auxiliaires du cache pour faire de la rétro-ingénierie des préchargeurs en déterminant leur effet sur le cache et pour obtenir**

**des informations précieuses sur leur fonctionnement ?**

Cet objectif général peut être divisé en plusieurs questions de recherche :

- (**Q1**) Comment déterminer quel est l'effet exact d'un préchargeur sur le cache en réponse à une séquence donnée d'accès à la mémoire ?
- (**Q2**) Quelle primitive de cache est plus appropriée pour la rétro-ingénierie des préchargeurs ?

D'autres questions sont apparues lors de notre enquête sur Flush+Flush :

- (**Q3**) Comment pouvons-nous expliquer et modéliser les variations des temps d'exécution de `clflush` ?
- (**Q4**) Comment peut-on améliorer la fiabilité de Flush+Flush pour construire une primitive aussi fiable que Flush+Reload ?

## 3. Contributions

Ce travail de recherche a donné lieu à deux publications : *Calibration done Right : Noiseless Flush+Flush*, publié à DIMVA 2021 [26], et *Characterizing Prefetchers using CacheObserver*, publié à SBAC-PAD 2022 [27].

### 3.1. Calibration done Right : Noiseless Flush+Flush

(Littéralement : *Une calibration faite correctement : Flush+Flush sans bruit*)

Ce premier axe de recherche a suivi le constat que Flush+Flush était encore moins fiable sur nos processeurs de 2019 que sur les processeurs utilisés par l'article original [36].

Nous avons étudié la source exacte des variations du temps d'exécution et confirmé qu'elles étaient dues aux différentes tranches de cache, une hypothèse mentionnée dans l'article original. Les tranches de cache sont des subdivisions du cache de dernier niveau de ces processeurs, le plus grand des caches du système. Chaque tranche, située à côté d'un cœur, est responsable d'une fraction de l'espace d'adressage. Nous avons pu utiliser les variations temporelles pour déterminer la topologie de l'interconnexion et le chemin critique probable des demandes, où "chemin critique" désigne la chaine de requêtes la plus longue qui détermine le temps d'exécution. Ceci répond donc à la question **Q3**.

En utilisant ces connaissances, nous avons pu améliorer de manière significative Flush+Flush, en implémentant un algorithme de calibration qui prend en compte la structure du cache en tranches pour réduire le taux d'erreur. Nous avons effectué des mesures de performance à la fois dans une configuration de canal caché et dans une configuration d'attaque par canal auxiliaire et avons ainsi répondu à la question **Q4**. Notre primitive Flush+Flush améliorée compense un bruit légèrement supérieur à celui de Flush+Reload par sa plus grande vitesse, ce qui se traduit par une bande passante dans notre benchmark de $5.81\,\mathrm{Mbit/s}$ pour Flush+Flush par rapport à $5.57\,\mathrm{Mbit/s}$ pour Flush+Reload. Les propriétés de Flush+Flush signifient qu'il peut être utilisé de manière réaliste pour surveiller une plage d'adresses même si les préchargeurs sont activés dans le système, ce qui montre que l'approche consistant à surveiller une plage d'adresses est une solution possible à la question **Q1**.

Comme effet de bord de cette recherche, nous avons dû appliquer une technique pour déterminer comment l'espace d'adressage est réparti entre les différentes tranches. Nous avons utilisé la technique publiée par Clémentine Maurice et al. [67] à un processeur dont la fonction d'adressage des tranches n'avait pas été identifiée précédemment. Les fonctions d'adressage résultantes sont présentées dans l'Annexe A.

Pour résumer, avec cet article, nous présentons les contributions suivantes :

- l'identification de la topologie de l'interconnexion présente dans les processeurs Intel dérivant de l'architecture Skylake client,
- la modélisation du timing de `clflush`, en fonction de la topologie, sa principale source de variations,
- un algorithme de calibration tenant compte de la topologie qui permet de construire une primitive Flush+Flush rapide et fiable.

## 3.2. Characterizing Prefetchers using CacheObserver

(Littéralement : *Caractérisation de préchargeurs à l'aide de CacheObserver*)

Après avoir construit un canal Flush+Flush fiable, ce deuxième article s'est attaqué à l'objectif initial de la thèse, en tentant de faire de la rétro-ingénierie de préchargeurs. Cela a commencé par la construction d'une boite à outil permettant de surveiller l'impact sur une région de la mémoire de diverses séquences d'accès, appelées motif d'accès (*Access Patterns*), en utilisant le canal Flush+Flush, ainsi que la technique moins puissante Flush+Reload.

En recueillant les résultats d'une série d'expériences, couvrant de manière assez exhaustive des motifs d'accès courts, nous avons observé que Flush+Flush permettait une surveillance plus efficace d'une grande région mémoire par rapport à Flush+Reload, avec une amélioration de la complexité linéaire par rapport à la taille de la région. De plus, il a été observé que l'utilisation de Flush+Reload rendait le *L2 Stream prefetcher*, notre principal objet d'étude, plus agressif que lorsque nous utilisions Flush+Flush. Nous pouvons répondre ainsi positivement à la question **Q2**.

A partir de nos expériences, nous obtenons des informations assez détaillées sur le comportement du préchargeurs, bien plus précises que les recherches précédentes. Cela fournit des informations importantes sur ces préchargeurs, répondant ainsi à la question **Q1**. Bien que nous n'ayons pas abouti à une rétro-ingénierie complète du préchargeur, nous fournissons une image beaucoup plus précise que ce qui était connu auparavant.

Un exemple particulièrement saisissant du comportement que nous avons découvert est que le *L2 Stream prefetcher* produit des requêtes vers le début de la même page lorsqu'il approche de sa fin (et réciproquement aux adresses de la fin de page lorsqu'une série descendante approche du début de la page). En d'autres termes, le préchargement s'enroule autour des pages, ce qui est peu susceptible de précharger des lignes utiles mais garantit également qu'aucune requête n'est émise qui traverserait une limite de domaine de sécurité.

Nous pouvons résumer ces contributions comme suit :

- Nous avons développé une boite à outils expérimentale basé sur Flush+Flush pour étudier les effets des préchargeurs.
- Nous avons identifié plusieurs comportements des préchargeurs L2 sur les processeurs Intel.

## 4. Artefacts

Nous avons publié la base de code utilisée dans chacun de nos articles, ainsi que la boite à outils complète combinant les deux bases de code et le code de support à même le métal utilisé dans les recherches préliminaires.

- Le code utilisé pour *Calibration done Right : Noiseless Flush+Flush* est disponible à l'adresse `https://github.com/MIAOUS-group/calibration-done-right` ; il comprend une implémentation générique d'un canal auxiliaire sur le cache avec une calibration automatisé, instanciée pour Flush+Reload et Flush+Flush, ainsi que le code d'évaluation pour une attaque sur AES par clair choisi ainsi que pour l'évaluation de canaux secrets.
- Le code de l'outil CacheObserver est disponible à l'adresse `https://github.com/MIAOUS-group/CacheObserver`.
- La boite à outils unifiée combinant les moyens d'investigation à même le métal (sans système d'exploitation), ainsi que le code des deux articles, est disponible à l'adresse `https://github.com/GuillaumeDIDIER/dendrobates-t-azureus`.
- `https://github.com/MIAOUS-group/slice-reverse` est une mise à jour du code publié par Maurice et al. [67] pour ajouter la prise en charge des processeurs Coffee Lake à 8 cœurs.

## 5. Aperçu du manuscrit

Ce manuscrit est divisé en deux parties ; la première couvre le contexte et les travaux connexes nécessaires à la compréhension de la thèse, tandis que la seconde partie développe mes contributions. La première partie est divisée en trois chapitres, tandis que la seconde contient quatre chapitres.

- Le Chapitre 1 commence par un rappel sur l'architecture des ordinateurs et détaille les concepts importants d'architecture et de micro-architecture nécessaires à la compréhension de cette thèse. Il consacre une attention particulière aux caches et se termine par une présentation des spécificités des processeurs Intel que nous avons étudiés.
- Le Chapitre 2 présente les préchargeurs matériels(*hardware prefetchers* en anglais), aborde les concepts généraux des préchargeurs et passe en revue les diverses conceptions académique et les informations que les industriels ont bien voulu divulguer liées à la classe de préchargeurs que nous avons étudiée. Ces connaissances de base ne sont nécessaires que pour le Chapitre 6.
- Le Chapitre 3 explore les domaines étroitement liés des attaques micro-architecturales et de la rétro-conception de micro-architectures. Il discute des concepts et les illustre avec divers exemples, en se concentrant plus particulièrement sur les exemples pertinents pour nos recherches, principalement liés aux mémoires caches.
- Le Chapitre 4 expose plus en détail les motivations de notre recherche et de notre approche. Il couvre les résultats préliminaires qui ont conduit à l'étude de `clflush` dans le chapitre suivant.

- Le Chapitre 5 développe notre contribution relative à l'étude de `clflush` afin de découvrir les sources de variabilité de son temps d'exécution. Il dévoile la topologie des interconnexions de ces processeurs et propose un modèle et un algorithme de calibration tenant compte de la topologie, ainsi que des tests de performance de ce canal Flush+Flush amélioré. Cela correspond à notre publication *Calibration done Right : Noiseless Flush+Flush* [26].
- Le Chapitre 6 présente notre travail sur l'utilisation de Flush+Flush pour la rétro-ingénierie des préchargeurs L2 sur les processeurs Intel, principalement le *L2 Stream prefetcher*, en construisant un outil pour surveiller l'effet des préchargeurs sur une plage de mémoire. Ce travail a été publié à SBAC-PAD 2022 sous le titre *Characterizing Prefetchers using CacheObserver* [27].
- Le Chapitre 7 explore plus en détail l'architecture de la boite à outils *dendrobates*, qui combine la base de code des deux articles et un support supplémentaire pour l'exécution à même le métal.
- Le chapitre de conclusion résume nos contributions et nos conclusions et explore leurs conséquences ainsi que les possibilités de recherches futures.

# Introduction

## 1. Context

### 1.1. Why reverse engineer computer processors?

According to a program stored in memory, computers process information and communicate with the outside world. This latter ability is also known as input/output. At the heart of these devices, the *Central Processing Units (CPUs)*, or *processors* do the actual processing. The modern CPUs we study are very large-scale integrated circuits, nowadays featuring billions or even tens of billions of nanometric electronic devices known as *transistors* on a single silicon chip. CPUs are often described as the brain of computers, even though computers are by no means intelligent. The aphorism "Computer make very fast, very accurate mistakes," originating on Usenet, probably sums up accurately the fact that computers will execute to the letter the often incorrect orders given to them by humans, but will do so in a very accurate and speedy way.

Despite the general knowledge of principles of such circuitry, companies that design state-of-the-art CPUs, like Intel, AMD, IBM, ARM, or Apple, are reluctant to reveal how their CPUs work or the techniques they use to boost their performance. Instead, they document how to use them but not what is happening under the hood. There are, however, quite a few reasons to peek under the hood.

First, the manufacturers' skills cannot be relied upon to provide security. These CPUs are at the heart of our computers and manipulate our data, which is, by definition, private and precious. Thus, it is imperative to assess the security of those systems without relying on their manufacturer. This requires both independently understanding how they work and verifying how they behave. Due to the complexity of such systems, it has been shown that unintentional mistakes are made regularly, introducing potential security issues [18, 45, 53, 62, 65, 129].

Additionally, having a precise understanding of CPUs is as valuable to whoever studies the performance of computer software and hardware as it is to the security researcher. The ability to accurately model the behavior of various sub-systems of a CPU can be essential to researchers proposing new techniques; understanding the hardware is necessary to attempt to track down a bottleneck in a workload [13].

One last case where reverse engineering is needed is when the manufacturer does not even document its products' interface, called the *architecture*. This is, for instance, the case of Apple Silicon, where the team writing drivers to allow Linux to run on these systems has to start by reverse engineering the interfaces to the hardware. In our case, however, we are interested in CPUs manufactured by Intel, whose architecture is documented [47]. However, the implementation of this interface, the *microarchitecture*, is mostly undocumented.

It is thus valuable to study CPUs and uncover the details not disclosed by manufacturers *reverse engineering* is the name given to such attempts at determining how a device or program

works without any access to its internal structure or internal documentation.

## 1.2. What are prefetchers ?

Modern CPUs are pretty complex; the microarchitecture contains a variety of subsystems. Critical among the components making modern CPUs fast are the memory *caches*, small and fast memory elements that speed up most accesses to the larger and slower main memory. The caches are part of the memory subsystem, which significantly impacts performance. Consequently, many techniques have been developed to improve the performance of the memory system. One of them is a class of components called *hardware prefetchers*, which attempts to anticipate memory requests to reduce the time needed to fulfill them. Belonging to the immersed part of the iceberg that is a CPU design, these prefetchers are barely disclosed and mostly undocumented.

To illustrate prefetching using a human-scale example, let's consider a stereotypical French cook, busy making, for instance, a *blanquette de veau* or a blueberry pie.[1] The ingredients must be obtained from, for example, the larder, the fridge, and the garden (or potager). All this back and forth to fetch ingredients takes time. Now imagine our cook has an assistant. They can tell the assistant to bring all the ingredients and lay them out on the table, which is closer at hand. The table can be compared to a cache. A clever assistant can guess the recipe the cook is cooking and gather ingredients before they are requested. This would be an example of prefetching and a tastier one than the data processed by computers.

## 1.3. Leaking information out of the microarchitecture

Studying those prefetchers is hampered by one major obstacle: Prefetchers operate on the CPU caches, which are part of the microarchitecture. Consequently, it is impossible to directly query the state of the caches and observe the prefetcher operation. However, the cache state makes a difference in execution time, and it is thus possible to obtain measurements that reveal part of the state of the cache. This method of deducing concealed information from the execution time of a program is an example of a broader class called *side channels*, where some private information leaks through physical consequences (e.g., noise, electromagnetic emissions, execution time) of how an algorithm is implemented, for instance.

To relate this with human experiences, let's assume my neighbor heats his house using a chimney, and we are in the middle of winter. I cannot see over the fence between the house, so I cannot know if my neighbor is at home or on vacation. However, during winter, I can observe what comes out of his chimney. If there is smoke, my neighbor likely is at home; if there has been no smoke for the entire day, my neighbor probably is away.

Now, this side-channel does not seem very harmful, neighborhood gossiping at first glance unless you happen to be planning a burglary. However, side channels can often be used for more nefarious purposes. A well-known example that most readers will likely have heard about is the obtention of a safe's combination by listing to the noises made while turning its input knobs. In computer security, many sources of leaks can exist, including execution time, power consumption, electromagnetic emissions, or noise. A *side-channel attack* occurs

[1] *c.f.* Appendix B

when an attacker exploits a side-channel to violate security guarantees and get access to some information she should not have.

This study was initiated in the context of a renewed scrutiny of microarchitectural security after a series of high-profile attacks [17, 18, 35, 54, 62], most of which belonging to the novel class of *transient execution attacks*. However, the field of microarchitecture security is much older, with major precursor work in the late 90s and early 2000s [11, 55, 90], presenting the first use of timing and cache attacks, generally targeted at cryptography algorithms. Cache side-channel attacks are microarchitectural attacks that provide information about the state of the cache. These attacks do not reveal the values in the cache but hint at which cache lines are cached. To deduce the action of prefetchers, we need information about the state of the CPU caches; hence we will use make use of such cache side channels, namely the Flush+Reload [135] and Flush+Flush [36] primitives, which apply to the CPU we study.

## 1.4. The Flush+Reload and Flush+Flush cache channels

The field of cache microarchitectural channels underwent extensive development in the 2010s, thanks to, notably, the publication of the Flush+Reload attack in 2014 [135]. It relies on the `clflush` instruction, which instructs the CPU to write back a *cache line* to the main memory and remove it from the caches.

*Cache lines* are fixed-size groups of data, the unit of data manipulated by caches. The `clflush` instruction had been introduced by Intel in 2000, and thus, the 2014 Flush+Reload attack affected a wide range of processors. In the Flush+Reload attack, a given cache line is flushed to the main memory and reloaded. How long it takes to reload lets out whether another process brought that line back in the cache in the meantime, as accesses from the main memory are much slower than access from the cache. This side channel requires that the two processes share some memory space, usually read-only; an assumption often verified for code and read-only data. After the measurement, the cache line can be flushed and reloaded again, repeating the process. This primitive has been a fundamental tool for the boom in microarchitectural research and is used by most high-profile attacks belonging to the transient execution field.

On Intel CPUs, the `clflush` instruction takes a different time to execute depending on whether a line is present in the cache (cached) or not, which led to the Flush+Flush primitive [36], discovered in 2016. Compared with Flush+Reload, both the timed load and the following `clflush` are replaced in this attack with a single timed `clflush`. Under the same assumption as Flush+Reload, this primitive is faster, as the load has been removed, but noisier as the timing difference between `clflush` for cached lines (*cache hits*) and non-cached lines (*cache miss*) is much smaller. Unfortunately, the noise has increased with the rise in the core count of Intel CPUs since 2016.

Our kitchen metaphor can illustrate these variants: we now have several cooks, each with their own working space, and assistants who go between the main kitchen table (the cache) and the larder and fridge (the main memory). The main kitchen table is shared, but the cooks do not know precisely what is there; only the assistants see the table. When an assistant fetches an ingredient, it will take a longer or shorter time, depending on whether the ingredient is already on the table or not. This fetch time is the equivalent in the metaphor of the load time in Flush+Reload. If we assume that a cook can also instruct an assistant to put back something where it

belongs, on the ground of "I won't be using it anymore," which is the equivalent of a `clflush`, all the ingredients for a Flush+Reload attack are present. Thus, a cook can figure out what other cooks are doing by watching how long the assistant takes to bring back each ingredient after ordering those ingredients removed from the table. For the Flush+Flush primitive, it makes sense that an assistant takes a different time to put back where it belongs an ingredient that's on the table compared to an ingredient that has yet to be brought on the table. The nosy cook thus repeatedly tells their assistant to store back ingredients and deduces which are in use.

## 1.5. State of the microarchitecture reverse engineering

Microarchitecture security requires a deep understanding of CPUs and has thus fostered significant progress in the field of microarchitecture reverse engineering. Many of these works have focused on the caches [34, 39, 48, 59, 60, 67, 85, 92, 123, 134, 137], a prime target for side-channel attacks with persistent state. However, other components have also been reverse engineered and become attack targets. The Rowhammer [53] physical problem, where bit-flips can be triggered in adjacent locations in DRAM memory, has led to efforts in reverse engineering DRAM [91]. Similarly, branch predictors have been reverse engineered [12, 31, 118], and the research community has also looked at the structures used to schedule instruction execution. All those efforts usually ended up with microarchitectural attacks.

The industry has sparsely disclosed information relative to prefetchers. Intel only provides names and brief overviews for its four prefetchers in its optimization manual [46]. While a few academic studies have covered the L1 stride prefetcher [19, 39, 105], a single study has looked at the L2 Stream prefetcher [94]. They used Flush+Reload to obtain a first set of reverse engineering results sufficient to build an attack, even though they did not model the prefetcher fully. One fundamental challenge in all these studies is that they usually use load instructions as a measurement primitive, which may influence the prefetcher and limits the amount of data that can be extracted from a single experiment.

## 1.6. How do we use cache side channels to reverse engineer prefetchers

Now, how can these side channels help us reverse engineer prefetchers? In our metaphor, the prefetcher is a clever assistant who, in response to the series of ingredients requested by the cook, brings a few extra ingredients to the table, as they are likely to be needed soon. So, in our kitchen, the cook could try to test how his assistant handles this kind of prefetch by requesting ingredients and then finding out what is on the table. Except he cannot look at the table. If he uses a Flush+Reload technique, measuring the time the assistant takes to fetch some ingredients, he will make more requests which the assistant will act upon, likely prefetching more ingredients. Similarly, timing memory accesses to determine what is in the cache will influence the prefetcher and may change the experience results. Flush+Reload is thus challenging to exploit to reverse engineer prefetchers.

However, because `clflush` is an uncommon instruction, we assume that prefetchers might not react to it, and thus might be a more suitable tool to figure out how they work. This is thus the chief goal of our research: *Attempting to use Flush+Flush to identify reliably how prefetchers react to requests.* In the kitchen metaphor, the idea is to request a few ingredients, then tell the

assistant to put back every single ingredient, taking note of which one takes longer to put back, to figure out what additional ingredients have been prefetched.

To specify our investigation's scope more precisely, we investigate the microarchitecture of high-performance Intel CPUs, released around 2019, derived from the Skylake microarchitecture. We focus our studies on the cache system, with prefetchers as the primary target.

## 2. Goals

The overall goal that drove the line of research can be summarized as follows:

**How can we use cache side channels to reverse engineer prefetchers by determining their effect on the cache and to gain valuable insight into how they operate?**

This general objective can be divided into several research questions:

- (**Q1**) *How can we determine what is the exact effect of the prefetcher on the cache in reaction to a given sequence of memory accesses?*
- (**Q2**) *What cache primitive is more appropriate to reverse engineering prefetchers?*

Additional questions that arose from our investigation of Flush+Flush are:

- (**Q3**) *How can we explain and model the variations of* `clflush` *execution times?*
- (**Q4**) *How can Flush+Flush reliability be improved to build a channel as reliable as Flush+Reload?*

## 3. Contributions

This research work has resulted in two papers: *Calibration done Right: Noiseless Flush+Flush*, published at DIMVA 2021 [26], and *Characterizing Prefetchers using CacheObserver*, published SBAC-PAD 2022 [27].

### 3.1. Calibration done Right: Noiseless Flush+Flush (Q3 and Q4)

This line of research was caused by the observation that Flush+Flush was even more unreliable on our CPUs from 2019 than on the CPUs used by the original paper [36].

We investigated the exact source of the variations and confirmed that they were due to the different cache slices, a hypothesis mentioned in the original paper. Cache slices are subdivisions of the last-level cache in those CPUs, which is the largest cache in the system. Each slice, located alongside a core, is responsible for a fraction of the address space. *We were able to use the timing variations to determine the interconnect topology and the likely critical path for requests.* Where "critical path" designates the flow of requests that is the longest and determines how long a given instruction will take. This answers **Q3**.

Using this knowledge, we were able to improve significantly Flush+Flush, by implementing a calibration algorithm that considers the sliced cache structure to reduce the error rate. We ran benchmarks in both covert and side-channel settings and thus answered **Q4**. Our Flush+Flush primitive compensates for a slightly higher noisiness compared to Flush+Reload with its greater speed, resulting in a bandwidth in our benchmark of $5.81\,\mathrm{Mbit/s}$ for Flush+Flush compared to

5.57 Mbit/s for Flush+Reload. The properties of Flush+Flush means it can be realistically used to monitor a range of addresses even with prefetcher enabled in the system, showing that the monitoring an address range approach is a possible solution to **Q1**.

As a side effect of this research, we had to apply a technique to determine how the address space is mapped onto the various slices published by Clémentine Maurice [67] to a CPU that had not been mapped before. The resulting mapping functions are presented in Appendix A.

To summarize, with this paper, we present the following contributions:

- the identification of the topology used in Intel CPUs deriving from the client Skylake microarchitecture
- the modelization of `clflush` timing, depending on the topology, its main source of variation
- a topology-aware calibration algorithm that enables fast and reliable Flush+Flush cache channels

## 3.2. Characterizing Prefetchers using CacheObserver (Q1 and Q2)

After having built a reliable Flush+Flush channel, this second paper tackled the initial goal of this Ph.D., attempting to reverse engineer prefetchers. This started with building a framework that could monitor the impact on a memory region of various sequences of accesses, called *access patterns*, using the Flush+Flush channel, alongside the less powerful Flush+Reload technique.

Collecting the results of a series of experiments, covering pretty exhaustively short access patterns, we observed that Flush+Flush permitted more efficient monitoring of a large memory region compared to Flush+Reload, with an improvement in complexity linear in the size of the region. Additionally, it was observed that using Flush+Reload made the main prefetcher we studied, the L2 Stream prefetcher, more aggressive than when we used Flush+Flush. We can thus answer **Q2** positively.

From our experiments, we obtain pretty detailed information on the behavior of the prefetcher and uncover far more precise information than the previous research. This provides significant insight on those prefetchers, answering **Q1**. Although we do not have a complete prefetcher reverse engineering, we provide a far more precise picture than was previously known.

One particularly illustrative example of behavior we uncovered is that the stream prefetcher will issue prefetches to the start of the same page when nearing its end (and reciprocally when a stream towards lower addresses approaches the beginning of the page). In other words, prefetch wraps around pages, which is unlikely to produce beneficial prefetch but also ensures no prefetch is issued across security domains.

We can summarize these contributions as follow:

- We developed an experimental framework based on Flush+Flush to study the effects of the prefetcher.
- We identified several behaviors of the L2 prefetchers on Intel CPUs.

## 4. Artifacts

We have released the code base used in each of our papers, alongside the complete framework combining both code bases and bare-metal support code used in preliminary investigations.

- The code used for *Calibration done Right: Noiseless Flush+Flush* is available at `https://github.com/MIAOUS-group/calibration-done-right`; it includes a generic implementation of a cache channel with automated calibration, with instantiations for Flush+Reload and Flush+Flush, along with benchmarking code for AES Chosen Plaintext attack and covert channel.
- The code for the CacheObserver framework is available at `https://github.com/MIAOUS-group/CacheObserver`.
- The unified repository containing bare-metal investigation support, alongside the code from both papers, is available at `https://gitlab.inria.fr/uarch/dendrobates-t-azureus` and `https://github.com/GuillaumeDIDIER/dendrobates-t-azureus`.
- `https://github.com/MIAOUS-group/slice-reverse` is an update to the code published by Maurice et al. [67] to add support for the 8-core Coffee Lake CPUs.

## 5. Outline

This manuscript is divided into two parts; the first covers the background and related work necessary to understand the thesis, while the second part develops my contributions. The first part is divided into three chapters, while the second one contains four chapters.

- Chapter 1 starts with a refresher on computer architecture and details the important architecture and microarchitecture concepts needed to understand this thesis. It includes a significant focus on caches and ends with a presentation of the specifics of the Intel CPUs we studied.
- Chapter 2 introduces hardware prefetchers, discusses general prefetcher concepts, and surveys various academic designs and industry disclosures related to the class of prefetchers we studied. This background knowledge is only required for Chapter 6.
- Chapter 3 explores the intertwined field of microarchitectural attacks and microarchitecture reverse engineering. It discusses the concepts and illustrates them with diverse examples, focusing more intensely on the examples relevant to the work, chiefly related to memory caches.
- Chapter 4 exposes more detailed motivation for our research and approach. It covers preliminary results that led to the investigation of `clflush` in the following chapter.
- Chapter 5 develops our contribution related to the study of `clflush` in order to uncover the sources of variability in its execution time. It uncovers the topology of those CPUs' interconnects and proposes a model and a topology-aware calibration algorithm, along with benchmarks of this improved Flush+Flush channel. This corresponds to our work *Calibration done Right: Noiseless Flush+Flush* [26].

- Chapter 6 presents our work on using Flush+Flush to reverse engineer L2 prefetchers on Intel CPUs, chiefly the L2 Stream prefetcher, by building a framework to monitor the effect of prefetches on a memory range. This work was published in SBAC-PAD 2022 under the title *Characterizing Prefetchers using CacheObserver* [27].
- Chapter 7 explores in more detail the architecture of the dendrobates framework, which combines the code base of both papers and additional bare-metal support.
- The Concluding chapter summarizes our contributions and conclusions and explores their consequences along with potential further research.

> **Takeaway:** Throughout the manuscript, boxes like this one summarize or points out essential ideas that the reader should take away from a given section.

> **Takeaway:** In this introduction, we have seen a broad overview of prefetchers and how we plan to use cache side channels to study them, notably the Flush+Flush primitive. We then formalized our goals and introduced our contributions, the *Calibration done Right: Noiseless Flush+Flush* and *Characterizing Prefetchers using CacheObserver* papers. The former publication reverse engineers the topology of the interconnect in Intel CPUs and uses this knowledge to improve Flush+Flush reliability. The latter uses this improved Flush+Flush primitive to monitor a memory range for the effect of prefetchers and uses this to uncover new knowledge about the L2 prefetchers in Intel CPUs.

## Acknowledgments

This work has been partly funded by the French Direction Générale de l'Armement, and by the ANR-19-CE39-0007 MIAOUS. Some experiments presented in Chapter 5 were carried out using the Grid'5000 test-bed, 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`).

# Part I.

# Background

# 1. Computers, CPUs and Caches

This thesis is about hardware prefetchers, a mechanism found in modern computers as part of their processors. We will thus spend most of this thesis discussing minute details of computers' inner workings. It is only fair, dear reader, that I first explain to you how computers work, in particular their *Central Processing Units (CPUs)*, also known as *processors*.

We will first (Section 1.1) see how the CPU fits in a computer and introduce the CPU Architecture abstractions. Then we will dive into the architectural challenges involved in making a CPU fast (Section 1.2). From there, we will then focus on the caches (Section 1.3) used to solve one of the previous challenges and where lies the hardware prefetchers studied in this thesis. We will take a detour to discuss Operating Systems and how it interacts with CPU design (1.4) and, lastly, expose the specifics of the CPUs from Intel we studied (Section 1.5).

## 1.1. What is a Computer?

Hopefully, most readers will have already seen and used a computer, either as a tall rectangular, often black, box sitting under a desk or as a laptop, with a screen, a keyboard, and a mouse. These devices are both computers, and so are servers, sitting in data centers, horizontal slabs with neither screen nor keyboard anywhere close. As seen in Fig. 1.1, these devices are quite different from one another, yet all are called computers, as are many others. What is thus a computer?

### 1.1.1. Abstract view: The Von Neuman model

At an abstract level, all modern computers behave following a model, first formalized in 1945 by Von Neumann in a report formalizing the model used in the EDVAC project [80]. At its core, that computer contained a computing unit and a memory. It also possesses means of inputs and outputs. The computing unit is nowadays called the *Central Processing Unit* or *CPU*. It accesses the memory to read instructions, executes them one after the other, and reads and modifies data in its memory according to those instructions. It is worth noting that in this model, instructions are a kind of data and can thus be modified. Figure 1.2 represents this model.

> **Takeaway:** A computer is a device that executes instructions from its memory one after the other to read inputs, produce outputs, and modify its memory.

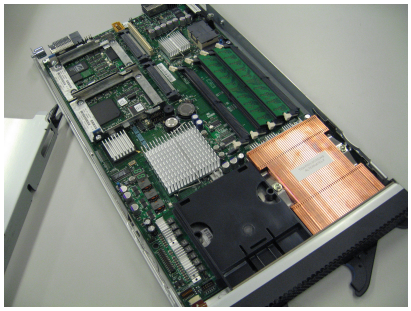
### 1.1.2. the Instruction Set Architecture (ISA) and the Microarchitecture

#### The contract between the programmer and the CPU manufacturer

Computers execute programs of instructions, but what do instructions look like?

(a) A workstation

(b) A laptop



(c) A blade server

Figure 1.1.: Photograph of various examples of computers: A workstation (left), a laptop (center) and a blade server (right)
Photograph of the IBM HS20 blade server (left) by Robert Kloosterhuis, licensed under CC-BY-SA.

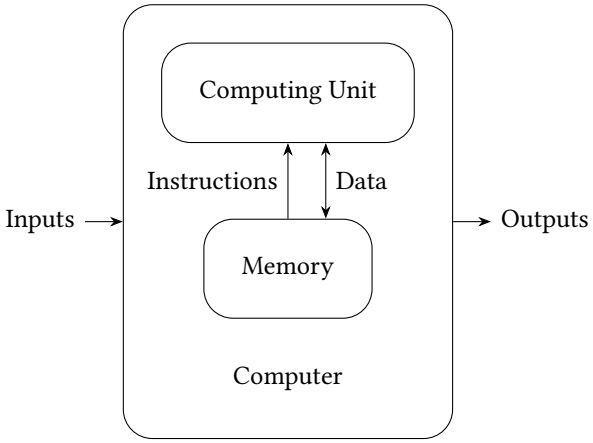



Figure 1.2.: Von Neumann architecture

*Machine instructions* are usually simple operations such as computing a simple mathematical operation, checking a condition, reading or writing in the computer memory or local storage called registers, checking a condition, and diverting the flow of execution. Some CPUs have more complex instructions than others. The design of the set of instructions understood by a CPU is a discipline of its own and not within the scope of this thesis. We will refer the reader to a book such as *Computer Architecture - A Quantitative Approach* [41] — known as the *Hennessy & Patterson*, for discussion of the topic and the distinction between Reduced Instruction Set Computers (RISC) and Complex Instruction Set Computers (CISC).

Instructions can be classified into the following categories: First, the *arithmetic instructions* will compute mathematical operations such as addition, subtraction, division, or square root. We distinguish the *integer* arithmetic instructions, operating on a subset of relative numbers, from the *floating-point* arithmetic instructions, computing on a rough approximation of rational numbers[1]. Next are the *memory* instructions, that either *read* from memory, or *write* to memory. Last, the *branch* or *jump* instructions redirect the flow of execution, either conditionally or unconditionally. Overall, machines manipulate numbers represented using a finite number of binary digits (or bits). Instructions themselves are also encoded in binary and occupy one or several bytes (groups of 8 bits) in memory.

Branch instructions can be divided into various subcategories. One criterion is whether the destination is encoded in the instruction or obtained from program data (register or memory), and another is whether the diversion of control flow is conditional or unconditional. In addition, dedicated instructions may simplify calling function calls and returning from such calls. Typically CPUs feature the following types of branches:

- *Conditional (direct) jumps* are, stricto sensu, the only *branching* instructions; they divert the control flow depending on a run-time condition. If this condition is verified, the next instruction is a destination encoded in the instruction; otherwise, it is the next instruction.
- *Unconditional (direct) jumps* always divert execution flow to an address encoded in the instruction.
- *(Unconditional) Indirect jumps* divert the execution to an address stored in a specific location. This is generally a register, but could also be a memory address.
- *Call instructions* are a specific kind of unconditional jump that records the address of the next instruction in an appropriate location so as to return to it at the end of the routine. Typical locations to record this may be a register or data memory.
- *Return instructions* jump back to an address recorded by a call instruction; they generally read back from the location call instructions and store the return address.

The precise set of instructions and their complexity depends on the processor design. However, as not all machines use the same instruction set and instruction encoding, or *machine language*, a different binary program is needed for each such language. In those times when Von Neumann wrote, computers were few and purpose-built, and each model tended to have its own machine

[1] The details of number representation, including IEEE 754 floating point, is out of scope in this thesis. We refer the reader to a book such as *Computer Systems: A Programmer's Perspective* (CS:APP) [16] for more details

language. However, nowadays, with millions of computers built each year, with many different models, it would be intractable to deal with each model's unique flavor of machine language.

One could have standardized a specific model of CPU and required all computers to use it, but this would have prevented manufacturers from moving forward and making faster CPUs. Thus, some kind of abstraction is required, a contract between the hardware manufacturer and the programmer that defines an abstract model of the machine the programmer can rely on and leaves room for the manufacturer to implement in evolving designs.

The *Instuction Set Architecture (shortened to ISA or Architecture)* defines a machine language and programming model that many CPUs can implement. The internal organization of CPUs is called the *microarchitecture* and is allowed to change from CPU to CPU. A program targeting a given ISA is thus portable across all CPUs implementing this ISA.

> **Takeaway:** The Instruction Set *Architecture* (ISA) is the contract a programmer can rely on to make a program that runs on many CPUs *microarchitectures* implementing this architecture.

#### Machine programming, higher-level languages, and compilers.

Binary encoding of machine instruction is well suited to machines but not so much for humans, as shown on the right of Fig. 1.3. One would be hard-pressed to guess the purpose of the code from its machine code. While the first programs were translated into machine code by hand, it was a tedious and error-prone process, and people quickly made use of computers to simplify the process.

The first step was to define textual names for the instructions and write a program to translate the textual representation into the binary one. This program is called the *assembler*, and the primitive language it defines is called the *assembly* language. Each architecture has its assembler, and sometimes different assemblers exist, possibly using different conventions. The x86 architecture we studied features two assembly syntaxes, AT&T's and Intel's, and we exclusively use, in the thesis, the AT&T syntax. Disassembling a binary usually gives a result pretty close to the original assembly. The assembly corresponding to the factorial machine code can be found in the middle of Fig. 1.3.

While this is better, each assembly syntax is architecture-specific and often overly verbose. This led to the development of higher-level languages, further abstracted away from the specificity of the architecture. These languages have the same syntax on every architecture and are translated first into assembly and then binary by another program, the compiler. In some cases, the same high-level code will compile on multiple architectures.

These languages can be a lot more independent from the architecture and allow code to be written to work on several platforms, with the compiler handling the specifics of the architecture. However, programs are still compiled for a specific architecture, and recompilation is needed to run on a different architecture. As many companies insist on providing their program only as compiled binaries, it is essential to retain a compatible architecture. Portable code can be written in higher-level languages, but most code is not written in this way. Consequently, recompiling for a different architecture generally requires small changes in the *source code* written in the higher-level language. This increases the need to retain ISA compatibility.

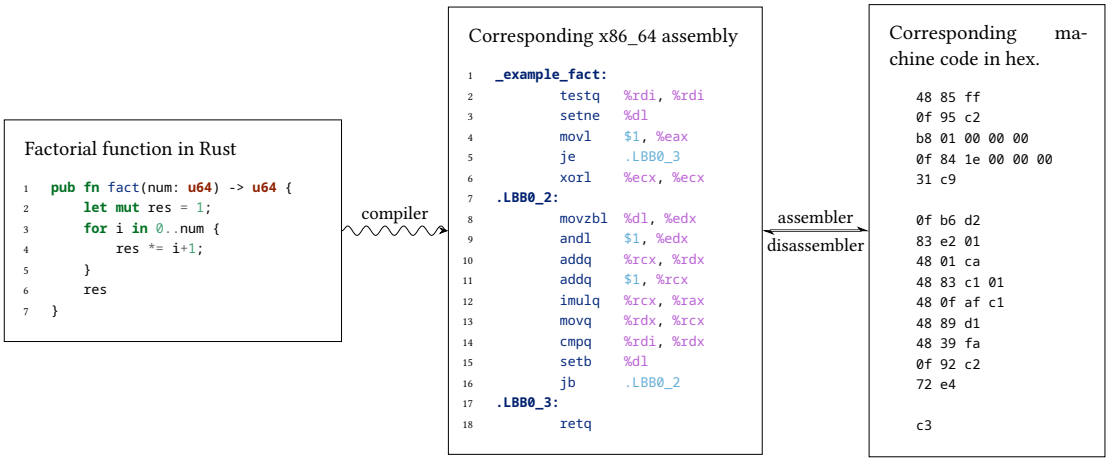



Figure 1.3.: A factorial function in Rust compiled to assembly and assembled to machine code. The assembly and machine code are obtained using rustc 1.62.0 with `-C opt-level=1`. The machine code bytes have been split to correspond with the assembly.

Many higher-level languages exist, such as the C programming language, the OCaml programming language, or the Rust programming language, the study of which is outside the scope of this work. We used programming language as a tool; this work mostly used the Rust programming language along with a little handwritten assembly. One can find the much more understandable code for the factorial function on the left of Fig. 1.3.

> **Takeaway:** Programs are usually written in higher-level languages, where a compiler translates the code to the specific machine code for the target ISA. Binary portability still requires architecture compatibility.

In this section, we have thus seen the definition of a computer and a CPU, the distinction between architecture and microarchitecture, and an overview of how machines are programmed. In the next section, we will dive into what lies below the architecture, into the microarchitecture.

## 1.2. Building fast CPUs

While a CPU has to execute programs correctly, the goal is also to make this execution as fast as possible. We will thus have a glance at the technology used to build CPUs, look at the constraints that the CPU architect faces, and what solutions have been found to make faster CPUs.

### 1.2.1. How are CPUs made?

CPUs are physical devices, and thus building them requires dealing with the physics of electronic devices. At this time, CPUs are generally manufactured using the complementary metal oxide semiconductor technology, or CMOS. The details of this technology are not within the scope of

this thesis. We will refer the interested reader to a reference book such as *CMOS VLSI Design: A Circuits and Systems Perspective* [128]. For our purpose here, it is enough to understand that these circuits are built using (MOS) transistors arranged in logical gates. These gates can be used to build complex logic functions and memory elements. One notable memory element is the *hardware register* or *flip-flop* (we will use the latter word to avoid confusion with the architectural registers, which are hardly ever built using flip-flops [128]).

*Flip-flops* sample their input on a rising edge of a clock and maintain their output to the value they sampled for a whole cycle. From flip-flops and logic gates, circuits can be built that will execute an instruction every clock cycle, given a clock signal.

The CMOS technology has seen yearly refinements and improvements over the past decades, with the ability to profitably manufacture smaller and smaller transistors in increasing numbers, with Gordon Moore showing in 1965 that the number of transistors it was profitable to put on a chip doubled every two years [77]. This trend brought to light in the 60s has held until the 2010s, at which point the rate of increase slowed down a bit below Moore's prediction, facing physical limits.

> **Takeaway:** A major trend in computer architecture is an exponential increase in the transistor budget over time.

### 1.2.2. The Critical Path, aka why a simple CPU cannot be fast.

However, the design we just described above has one major issue. Logic gates, made of transistors, do not react instantly; thus there is a delay between the time when the input reaches its new value and the time when the gate output reaches its final value. When one considers a logic function, the maximal delay between an input and an output is called the *critical path*, and this path constrains the clock frequency; the circuit may not be run at a higher frequency and be guaranteed to operate correctly.

As a result of the delays required to execute an entire instruction, fetching it from memory, decoding it, executing it, and writing back the results; the "one cycle - one instruction" design above is kneecapped by its critical path and can only operate at a very low frequency.

> **Takeaway:** More sophisticated techniques must be used to make high-performance CPUs, generally attempting to leverage a larger number of transistors to work around-the-clock frequency limitations, as we will see in the coming subsections.

### 1.2.3. Pipelining, throughput vs latency

One major insight is that performance is often related to executing many instructions in a given time, not necessarily by how fast a single instruction executes. The *pipelining* technique consists in splitting execution into several smaller stages and having an instruction executing in each stage. Thus several instructions progress simultaneously through the system.

**Latency** is defined as the time it takes from the point at which a given instruction starts executing to the time its execution is complete.

**Throughput** is defined as how many instructions can be processed in a given slice of time.

When splitting the logic of a CPU into several stages, the latency (in second) increases as extra flip-flops must be inserted, adding delays, but the total throughput is improved. If the instruction stream only contains independent instructions, we can express the theoretical throughput (in instruction per second) and latency (in seconds) of an $n$-stage pipeline depending on the latencies (in seconds) of the flip-flops, $L_{flipflop}$ and of the critical path excluding the flip-flops, $L_0$. The latency is $L = L_0 + n \times L_{flipflop}$ and the throughput $T = \frac{n}{L} = \frac{n}{L_0 + n \times L_{flipflop}}$.

However, instructions in real workloads generally include dependencies in data, when instructions use results of previous instructions as inputs, and control flow, as *branch* instructions change what instruction is the next to be executed depending on computation results. Consequently, the real throughput also depends on the latency added by those dependencies, which increases with the number of pipeline stages.

Consequently, it is not possible to indefinitely increase the number of stages of a pipeline, and different techniques must be used to actually build a fast CPU.

The key insight to remember here, however, is that to make a CPU fast, one must find ways of executing things in parallel. *Parallelism* is how pipelining improves throughput despite higher latency, and parallelism is how modern CPUs use increasing numbers of transistors to increase their throughput.

Consequently, with pipelining having reached its limit, the next idea was to find ways of executing instructions in parallel, to execute more than one instruction per cycle. Such a CPU is called *superscalar*.

> **Takeaway:** With increasing numbers of transistors, modern CPUs must find ways of executing as much work in parallel. *Pipelining* is the fundamental idea of splitting execution into several smaller steps. However, it is limited by instruction dependencies. Further development aimed at *superscalar execution*, where parallel execution leads to an average execution of more than one instruction per cycle.

Building such a Superscalar CPU involves three different challenges tackled by Modern CPU design.

### 1.2.4. Modern CPU problems: Feeding the pipeline

To execute many instructions in parallel, one must identify where execution is going, that is what will be the next instructions in order to feed the pipeline of the system.

On average, a branch instruction occurs every five to ten instructions [41, 58]. In addition, other instructions can also divert the execution when encountering an error condition, also known as a *fault*. Examples include dividing by zero or accessing an invalid memory address. Thus, it is impossible to know for sure what the following instruction will be, let alone far enough in the future for hundreds of instructions to be in flight simultaneously in the CPU as is done in modern CPUs.

However, while CPUs cannot know for sure ahead of time, it has been shown that many branches are rather predictable, and most of the time, the instruction that could encounter a fault does not, in fact, encounter one. People mostly try to execute well-formed programs to obtain results instead of error messages about misbehaving programs.

The modern CPU pipeline thus starts with a Fetch stage that attempts to accurately guess where execution will go, records those guesses, and validates them once the instruction has been executed. The CPU is then structured to be able to throw away the state corresponding to erroneous guesses. This is an example of a more general design principle in computer architecture: predict, that is guess somewhat accurately, values that are needed before they are known [41, 89]. Many such values tend to be amenable to accurate predictions. Non-branch instructions are executed sequentially, so the fetch stage usually fetches the next cache line; only branches (including unconditional jumps) require specific handling.

To deal with branches, the fetch state includes components named *branch predictors*, tracking the branches and their past behaviors. There are three predictions they must make: (1) Is the current instruction a branch, conditional or unconditional? (2) If the current branch is conditional, will it be taken? And (3) what is the target of this branch instruction?

Many designs and techniques have been proposed to solve these three problems [41], but most modern CPUs rely on designs derived from TAGE [101], with accuracies over 97% [101, 102]. This predictor is an instance of the more general concept of hybrid predictors, which integrates prediction from various predictors and selects the one most likely to be accurate.

Returning from procedure calls is a type of branch that is difficult to predict accurately with conventional techniques. However, it is amenable to prediction when looking at the sequence of call instructions. A structure dedicated to tracking calls and returns, often called the *Return Address Stack (RAS)* is often included in modern CPUs' fetch stage. When a call is made, return addresses are pushed onto the stack and popped as the next instruction prediction when a return instruction is encountered.

> **Takeaway:** In modern CPUs, branch predictors keep the pipeline fed by accurately anticipating the next instructions that will be executed. The Return Address Stack (RAS) is used to deal with the unpredictable return instructions.

### 1.2.5. Modern CPU problems: Keeping the core busy

After identifying and decoding the stream of instructions to be executed, the next challenge is to find ways to execute in parallel as many instructions as possible. Modern CPUs feature many execution units. These units can handle different kinds of instructions, some units deal with simple arithmetic instructions, some deal with multiplications and divisions, some deal with floating-point instructions, and others deal with memory for instance. CPUs may contain several units capable of executing frequent instructions while more exotic instructions are sometimes supported only by a single unit.

The pipeline is kept busy by identifying instruction dependencies and executing instructions as soon as an appropriate unit is available and the dependencies are resolved. One key feature is that, at this point, instructions are not necessarily executed in program order; instructions with no dependencies can skip ahead of a line of instruction with a complex dependency chain if they execute on a different unit. This technique was first introduced in 1967, known as Tomasulo's algorithm [116], before being generalized in the 90s to microprocessors [138].

Different solutions exist to track instruction dependencies and to find eligible instructions to be executed on a given unit. We will refer the reader to reference computer architecture

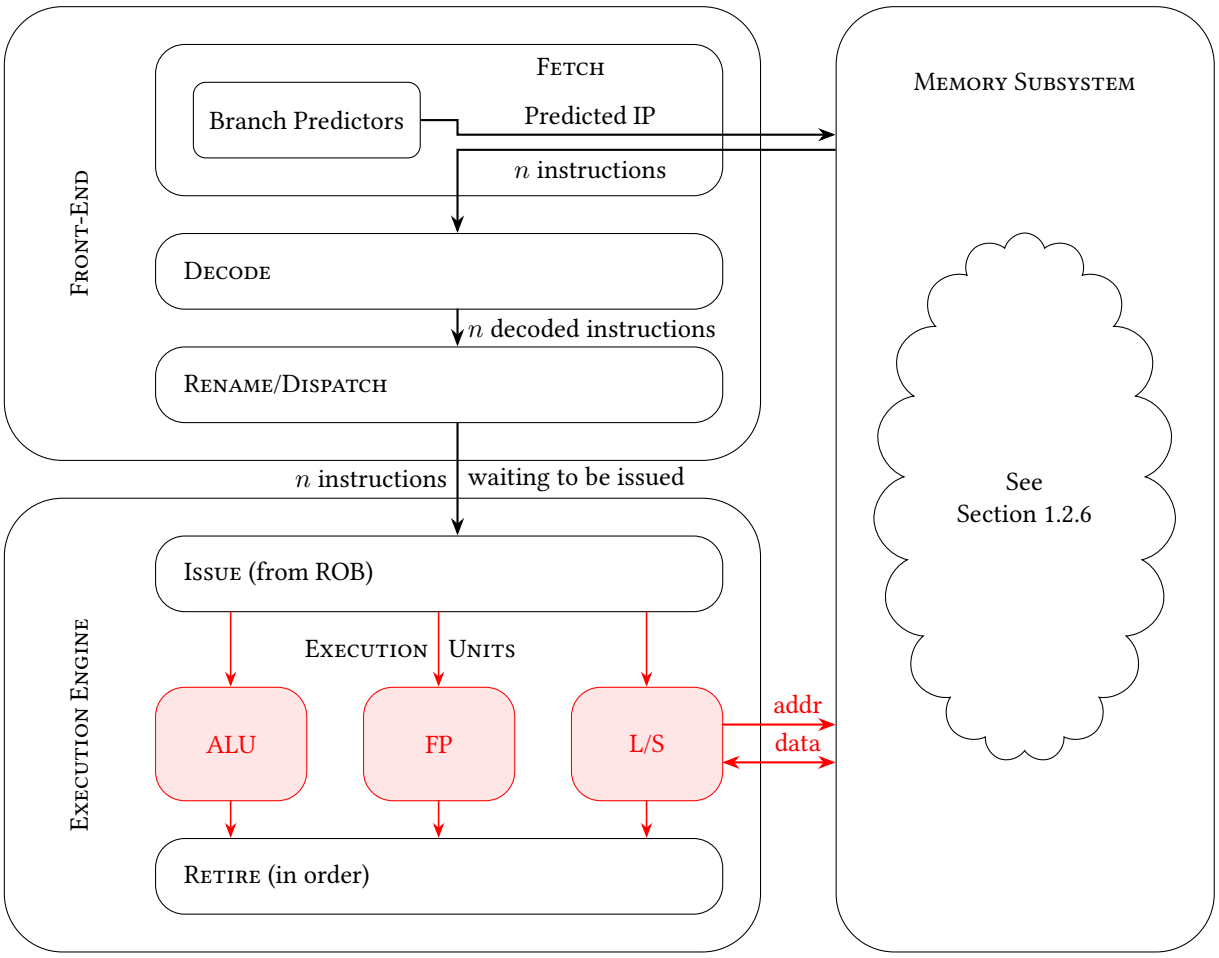



Figure 1.4.: An out of order pipeline, with an $n$ wide pipeline. In red the part where instructions are processed out of order.

books such as *Computer Architecture - A Quantitative Approach* [41] — known as the *Hennessy & Patterson* — and individual reports by CPU manufacturers [28, 30, 57, 107, 115, 138]. In addition, the terminology used depends on the CPU manufacturer.

One key feature, however, is that the CPU still tracks the original order of the instruction stream in a structure called by some manufacturers a Reorder Buffer (or ROB). The CPU uses this structure to retire instructions in order, validating that the in-order execution of the instructions has been completed without error and freeing up the resources used by those instructions. This step ensures that the in-order semantics of the architecture are maintained and that faults are handled precisely. Because most instructions in-flight are *speculative*, depending on the proper execution of the previous instruction and correct guesses from the branch predictors, this step is essential. The *retire* stage is when instructions stop being speculative.

Figure 1.4 shows an example Out of Order pipeline, in a cycle where two instructions are dispatched into queues (one to an Arithmetic and Logic Unit (ALU), and one to a Floating Point unit), one instructions is issued and starts executing, and two instruction retire, after a memory instruction finishes executing.

> **Takeaway:** To execute as much work in parallel as possible, CPUs spread execution over several execution units, executing instruction speculatively and out of order, while using an in-order retire stage to maintain the semantics of in-order execution.

### 1.2.6. Modern CPU problems: Dealing with memory

Another trend in microelectronics technology is an increasing performance difference between CPU performance and main memory access latency, called the *Memory Wall* [131]. Memory accesses can now take several hundred cycles to complete, which stalls the execution of all dependent instructions in the core. This evolution can be seen in Fig. 1.5. While out-of-order execution can hide some amount of latency by executing independent instructions while waiting for the memory access to complete, it is unable to hide latencies of a hundred cycles on instructions that may represent over 25% of the instruction mix [58].

The critical path constraints mentioned above result in a trade-off between memory speed and memory size. It is thus possible to build fast memories of a much smaller size and attempt to leverage those to speed up execution. Following the computer architecture principle ”Make the common case fast”, *cache memories* [42] are small and fast memories used to speed up memory accesses to frequently accessed memory locations, compared with main memory. Technology-wise, main memory is usually built using DRAM (Dynamic Random Access Memory) while caches are built using SRAM (Static RAM) [128]; however, the difference between these technologies has no direct impact on this work.

One key insight that makes this approach work is memory *locality*. Computer programs tend to re-use recently accessed data (*temporal locality*) and to use data stored close to data recently accessed (*spatial locality*) [89]. To leverage spatial locality and simplify cache organization (see Section 1.3.1), data is manipulated at the granularity of blocks of fixed size (for instance, 64 bytes), called *cache lines*.

It is worth noting that caching is a far more general concept and can be encountered in many fields, such as file systems or internet servers.

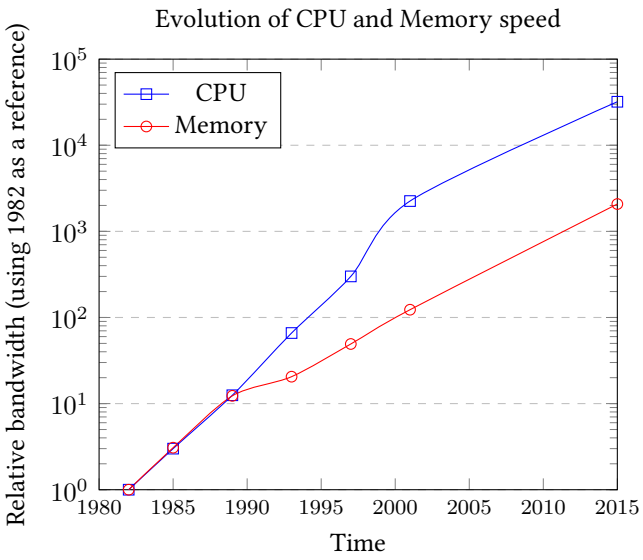



Figure 1.5.: Evolution of relative bandwidth of CPU and Memory, showing the Memory Wall, data from *Hennessy & Patterson* [41]

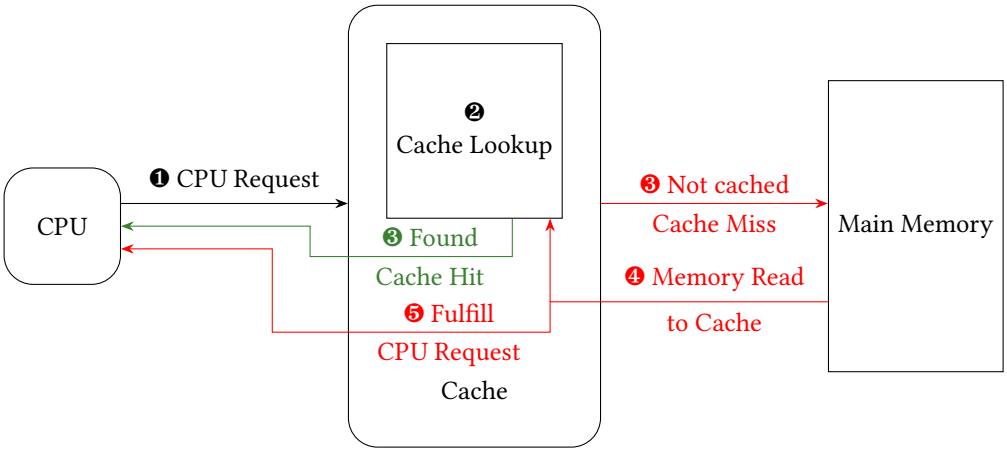



Figure 1.6.: Flow of a memory request through the cache

Figure 1.6 illustrate how a cache works. When the CPU makes a request (❶), the cache look-ups in its cache memory if it has a copy of the line containing the data requested (❷). If the line is found, the cache can answer immediately the CPU request (❸), in which case we say the request *hit* in the cache or encountered a *cache hit*. Otherwise, we say that a *cache miss* occurred, and the cache then requests the cache line from the main memory (❸), inserts the line in its cache memory (❹) and fulfills the CPU request (❺), a process that takes quite a bit longer than a cache hit.

> **Takeaway:** *Caches* are smaller and faster memories used to speed up most memory accesses and overcome the Memory Wall. Data is grouped in fixed-size blocks called cache lines. A *cache hit* occurs when a memory access finds the data it seeks in a cache, the opposite is a *cache miss*.

### 1.2.7. The limit of the superscalar core

Several obstacles have, however, hampered the single superscalar core. First of all, key structures in such cores scale quadratically with performance increases, hitting a point of diminishing return [82]. Additionally, faster cores consume an increasing amount of power, and the scaling of devices is now hitting limits. As a result device scaling can no longer reduce the power consumption of larger chips as it used to [78]. Faced with diminishing returns and increasing power constraints, one of the solutions used to increase the performance of systems was to ship multiple CPU cores on the same chip, using the transistor budget to improve at a better rate parallel workflows [82].

One adjacent technique to multicore is *Simultaneous Multithreading (SMT)*: to better exploit the execution backend of CPU cores, it is possible to share these resources to execute several flows of instructions at the same time. A single physical core behaves as several virtual threads of execution. Consumer CPUs generally have 2-way SMT, where a physical core executes up to two different threads of instructions, but large-scale 8-way SMT cores have been produced targeting specific server markets [114].

With power also being a new constraint, various additional techniques have been developed to manage and reduce the energy consumption of unused or underused resources in CPUs. For instance, power gating unused circuits and varying operating frequency and voltage depending on the workload and current power consumption have all been implemented across the board. The relation between the frequency and the minimum voltage it requires for operation imposes simultaneous variation of these parameters, hence the name *Dynamic Voltage and Frequency Scaling (DVFS)* [78]. DVFS can both be used to temporarily boost a single core speed when the power and temperature of the system permit it or to slow down cores when the system load is low.

> **Takeaway:** Limitations in scaling and power of superscalar processors have led to the development of multi-core CPUs, Simultaneous Multithreading (SMT), and the implementation of various power reduction techniques such as Dynamic Voltage and Frequency Scaling (DVFS).

An additional topic in computer system design is that of multi-socket systems. One can

package several CPUs in the same computer, doubling or quadrupling the number of cores in the system compared with a single multi-core CPU. Specific high-speed connections are necessary between these sockets, such as Intel QPI [41].

#### Summary

In this section, we have seen an overview of modern CPU design, with the increase in transistor budget and its use to parallelize execution, with pipelining, speculative out-of-order execution, fed by branch prediction, and use caches to speed up memory accesses. We have also seen more recent development to get around the limitation of this model, such as multicore systems and DVFS.

## 1.3. Cache systems in modern CPUs

Section 1.2.6 presents the general concept of CPU caches, which reduce memory access time. This section will delve into the finer details of the cache systems used in modern CPUs.

We will first look at how an individual cache is organized and managed and then at how a hierarchy of several different caches is used in a modern multicore CPU. Last, we will expose a classification of cache misses, which will also sum up most concepts in this section.

### 1.3.1. A modern Cache organization

In this section, we consider one individual cache. First, we recall that cache lines store a fixed number of consecutive bytes, e.g., 64 bytes, and the limits are aligned on 64 bytes boundaries. In practice, the lower 6 bits of an address are called the *offset* and identify a byte in a line, while the remaining higher bits are used to identify a cache line and will be called the *cache line address*.

Caches have finite capacity and store a fixed number of lines; we call *cache block* the slots in which a cache can store a line. A cache block contains the line data, some validity metadata, and a tag that identifies the line currently stored within.

#### Cache sets and finding data fast

Given a cache line address, we first need to determine if it exists in the cache and, if it does, where. One naive approach is to allow cache blocks to contain any line. In that case, however, locating a line requires checking every single block in parallel to see if the tag within matches the cache line address.

However, this approach, called *fully associative*, scales rather poorly and is thus only employed in specific cases and with small caches. Consequently, modern caches constrain where a given line can be stored so that finding whether a line is cached only requires checking a few blocks.

The opposite approach is the *direct-mapped cache*. Each line can only be mapped onto one block. This is usually determined by taking the lower bits of the cache line address to identify the block, and the tag is then checked to see if the block contains the line or another. However, direct-mapped caches are vulnerable to conflict, and performance collapses when a program uses two conflicting addresses.

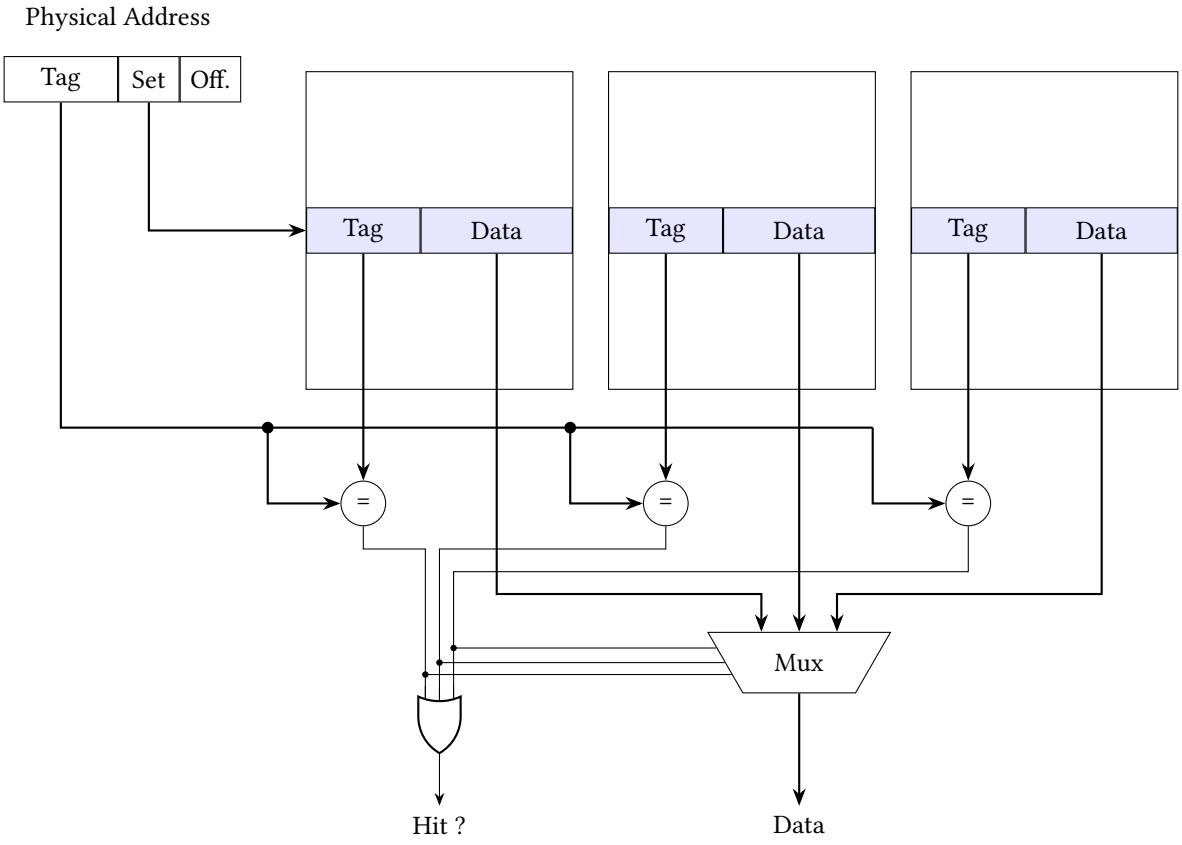



Figure 1.7.: Example of a 3-associative cache, illustrating cache sets

Most caches thus contain *sets* of equivalent blocks, where a given line is attributed to a set and can be stored in any of the blocks of the set. The number of blocks in a given set is called *associativity*. In particular, a *fully-associative* cache has an associativity equal to its number of lines, and a single set, while a *direct-mapped* cache has associativity 1 and its size in line is equal to the number of sets. Here again, a subset of bits of the cache address, usually the lowest, are used to identify a set. Typical associativity is often between 2 and a few dozen, while the number of sets can reach several thousand in the larger caches.

> **Takeaway:** Caches are usually organized in *sets*, where the address space is split equitably among the sets. *Associativity* is the number of blocks within a set, and is usually a small number, up to a few dozen at most.

### Eviction, what to do when the cache is full

We have seen how to find where a cache line is located. If it is in the cache, we can read the data and send it to the CPU. However, if it is not, we need to request it from memory before we can send it to the CPU. Consequently, the next question that arises is, should we insert this new

line in the cache, and where? More precisely, where should the line be placed within the set that this line belongs to?

Sometimes we have a block that contains no valid data in the set[2], but most of the time, all cache blocks contain valid data, and a victim must be picked to be *evicted* from the cache.

The *eviction policy* is the algorithm used to pick the line to be replaced in a given cache set when inserting a new line in the cache. The theoretically optimal policy is to remove the cache line whose next use is the furthest in the future [10, 42]; however, this is not a practical policy.

It is worth noting that eviction policy is a more general concept that is not simply restricted to hardware caches, as is caching. However, each application tends to have different constraints.

We will give here a few examples of eviction policies that are used in a significant number of caches.

**Random:** The victim is picked using a pseudo-random generator. This policy is documented as being used in CPUs from ARM, performs surprisingly well, and, as we will see later, is a hindrance to various microarchitectural attacks.

**Least Recently Used (LRU):** this policy tracks the uses of each one of the lines and picks the one whose last use is the furthest away in the past as the victim. This policy can behave poorly in certain pathological cases but generally has good performance. It may however be expensive to track precisely the uses.

**Pseudo-LRU:** As implementing LRU is expensive, policies that are close enough to LRU but simpler to implement in hardware are often used. They approximate LRU and retain most of the performance benefits.

**Most Recently Used (MRU):** This unusual policy may sometime be higher performing than LRU for certain patterns. For instance, repeatedly streaming through a long file that does not fit in the cache [20].

**Not Most Recently Used (NMRU):** This policy consists in selecting a line that is not the most recently used one. It is a coarse approximation of LRU. Confusingly, it is sometimes referred to as "MRU" in the literature, despite not evicting the most recently used line. This confusion may originate in the use of MRU bits to indicate lines that should not be evicted. [64, 76]

**Adaptative policies:** Caches may track the performance of various policies (for instance using each policy on certain sets) and adapt which policy they use on most sets, e.g., choosing between pseudo-LRU and NMRU [71, 123].

> **Takeaway:** The *eviction policy* determines which line in a set is removed to insert a new line (on a cache miss). *Pseudo Least Recently Used (Pseudo-LRU)* and *Random* are two policies commonly used in modern CPUs.

We have now seen in more detail how an individual cache is managed. This is but an overview of the field of caches, and many lower-level problems arise when building a high-performance cache. We refer the reader to books such as *Hennessy & Patterson* [41] and *CMOS VLSI Design: A Circuits and Systems Perspective* [128] or Alan Jay Smith's survey of *Cache Memories* [109].

[2] why this may arise will be seen in Section 1.3.2

### 1.3.2. Multiple caches within one computer

However, most systems do not contain, in fact, a single cache, but several. We will now look at both the reasons for this and how those caches work together.

#### The Cache hierarchy

The first reason for using multiple caches is similar to the reason for having caches in the first place. As expressed in *A Primer on Hardware Prefetching* [32] "Latency is traded off for capacity". Smaller memories are faster, larger memories are slower. Consequently, a multi-level hierarchy is built, with the lower-level caches being smaller and faster and higher-level caches, further away from the CPU, larger and slower than the lower levels but still faster than the main memory.

In this case, a significant portion of accesses is handled very fast, and then the higher-level caches handle a bit more slowly a fraction of the remaining access so that a very small fraction of accesses reach the main memory.

The Instruction and Data streams have different characteristics, and to simplify the parallel operation of the instruction fetch stage and the load/store units, it is common to have separate first-level caches for instruction (L1I) and data (L1D). These two first-level caches can then share the same unified L2 cache to handle misses in the first level. The L1s and L2 in Fig. 1.8 illustrates such a hierarchy.

> **Takeaway:** Modern CPUs use a *hierarchy* of caches, with lower-level caches being faster but smaller than the larger and slower higher-level ones. The first-level cache is often split into separate instruction and data caches.

#### Multi-core systems

We have seen earlier that modern systems include several cores on the same chip. The next question is thus: how do caches interact with multiple cores? There appear to be two options:

**Private caches** : Private caches are only accessed by one core. If several cores use the same data, a copy exists in each one of them.

**Shared cache** : Shared caches serve requests from several or all cores. They reduce duplication of data but may have bandwidth issues when attempting to scale to many cores.

Each of these options has trade-offs, but it is again possible to combine them. Thus modern systems often include private low-level caches and a shared higher-level cache. *Last-level cache (LLC)* designates this shared cache that is both the largest and the slowest in the system, shared by all the cores on a chip. Figure 1.8 presents a shared L3 cache acting as the last-level cache for private L1s and L2s.

> **Takeaway:** Multicore systems usually have a cache hierarchy with private low-level caches and a shared last-level cache.

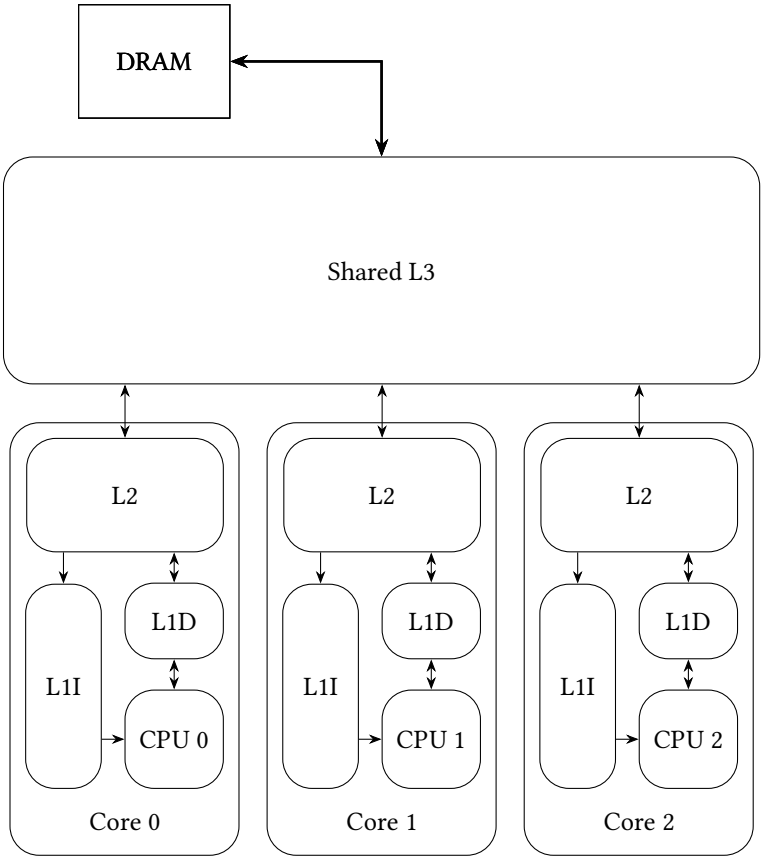



Figure 1.8.: An example cache hierarchy with instruction (L1I), data (L1D) and unified (L2) private caches per core, and a shared LLC (L3)

### Cache coherence

Now that we have several cores and several private and shared caches arises a new problem: *Cache coherence*. With several caches comes the potential that a line may be cached in several locations. A programming model where those locations can end up containing different values and the various cores can observe different sequences of values for the same memory address is quite difficult to use. Consequently, most architectures usually provide coherent caches, where it is guaranteed all the cores will observe the same sequence of values for a given cache line.

To implement such guarantees the caches implement a cache coherency protocol, which constrains the line location. The simplest protocol, named MSI, states a line can be in three states in each core: Invalid (I) when this core cache does not contain a valid copy of the line; Shared (S), when a core contains one of many unmodified copies of the line, and may freely read it, but not write it; and Modified (M) when the core has the only valid copy of the line in the system and may write it.

The private cache of a core that wishes to write must obtain the line in the M state to do so. If it is not already in this state, it must communicate with the other cores to invalidate their copies and ensure that it has the only copy latest value of the line in the *coherence domain*. When a core transitions away from the M state, it must write the modified value back to memory and transmit it to the core requesting the data. When transitioning to the I state it must also invalidate the line, and no longer possesses a copy.

The MSI protocol, and all the other protocols implemented by CPU manufacturers, ensure the Single Writer or Multiple Reader invariant (SWMR, read swimmer) on each line.

Protocols used in modern CPU are often extensions of the MESI protocol, described below:

**Invalid (I)** : The cache does not store a valid value, accesses are misses and require making a request to the next level.

**Shared (S)** : The cache holds a *clean* copy of the correct value, matching the one in memory, but other caches may also own one. The line can be read with no further request, but a write requires communicating with the other caches.

**Exclusive (E)** : The cache holds the correct value, as in the shared case, but it is additionally the only cache to do so. The line can be modified (and can transition to the Modified state) without any further request to the hierarchy.

**Modified (M)** : The cache holds exclusively a modified value. The stale value in memory must be updated before this *dirty* line can be evicted from the cache.

Figure 1.9 shows the state machine of the protocol. A Shared to Exclusive transition may be triggered if the ISA possesses a feature that allows programs to communicate their need for an exclusive access, for instance as an intent to write to the line at some point.

Implementation of this protocol can either be distributed, with coherence requests being broadcast between each core, also known as *bus snooping*, or Snoopy protocol; or can be implemented using a central directory tracking the coherency state of each line.

Cache coherence says nothing about ordering between different lines, this is the realm of memory consistency models, which is outside the scope of this work. We refer the reader to *A Primer on Memory Consistency and Cache Coherence* [79] for in-depth coverage of both topics.

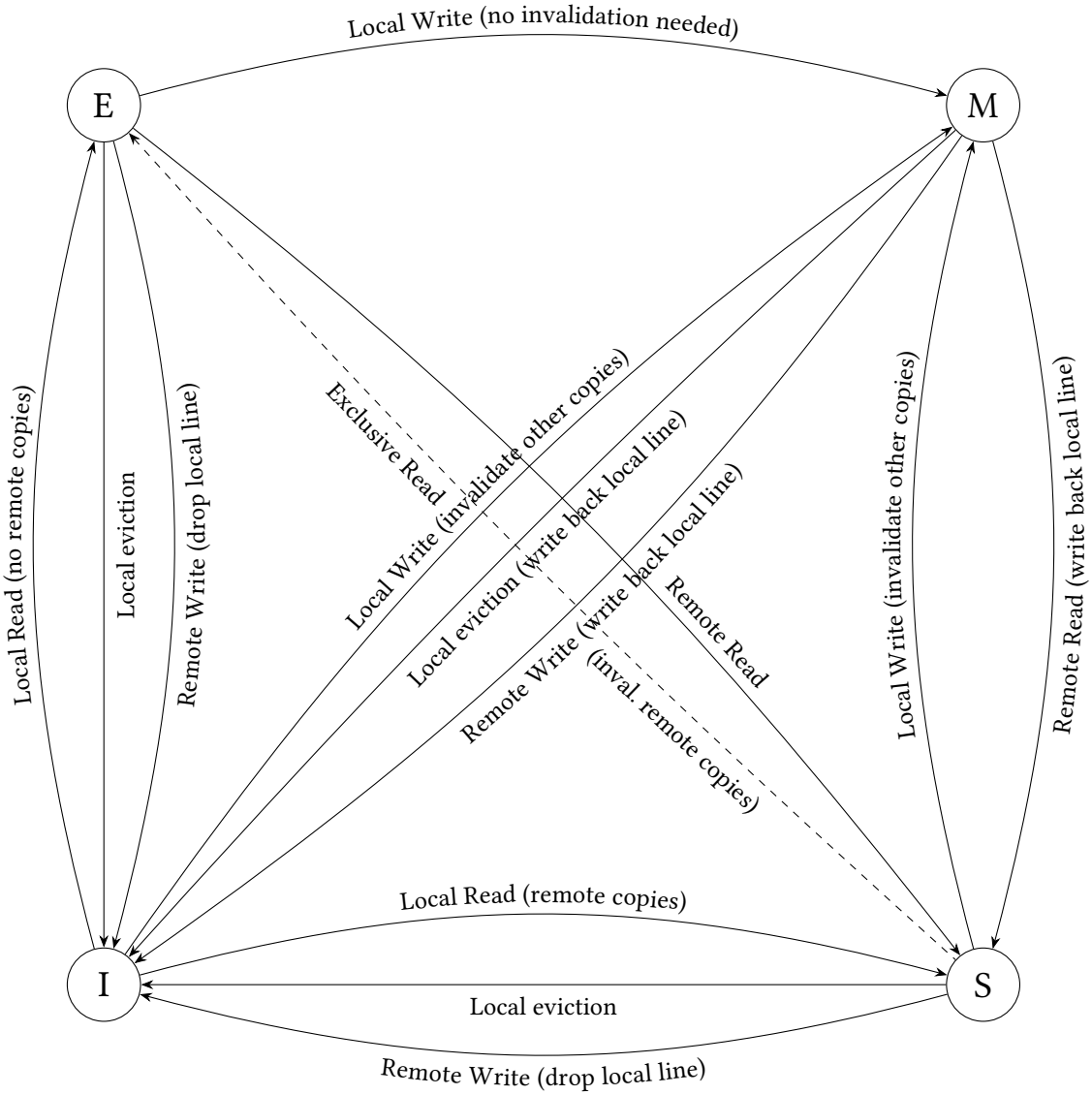



Figure 1.9.: State machine of the MESI protocol

> **Takeaway:** *Cache coherence* ensures that a given cache line has a well-defined sequence of values, on which all cores agree. It relies on a *cache coherence protocol* such as MESI to do so. It is to be distinguished from *Memory Consistency* that deals with the order different lines change in value may be observed by different cores.

#### Cache inclusivity

Another multiple cache management issue that arises too is that of *inclusivity*: Can a line be contained in both higher and lower level caches at the same time?

There are three possible policies in answer to this question :

**Exclusive caches** : In an exclusive cache, a line may not be present in two different levels.

**Inclusive caches** : In an inclusive cache, the higher level cache will include all the lines from the lower level caches.

**Neither inclusive nor exclusive** : In this case, lines from the lower-level and higher-level caches are managed independently by the eviction policies. Examples of such caches can be found in the AMD Opteron family at the L3 level [23, 40].

Having a Last Level Inclusive cache has the advantage of using that cache to store coherency meta-data along with the cache line itself, while non-inclusive last-level cache requires a separate directory for a centralized cache coherence approach, or a snooping protocol (with the negative bus traffic impact that arises).

In practice, real-world cache hierarchies may involve a mix of those policies, for instance, some Intel CPUs have an L3 inclusive of both L1s and the L2, with an L2 that is non-inclusive of the L1s [21].

Inclusive and Exclusive policies, can, as cache coherence, cause eviction in the lower caches to maintain the invariants, without replacing the evicted line in the lower cache.

> **Takeaway:** We call *inclusive* a cache that must contain all lines found in lower-level caches, and *exclusive* a cache that may not contain lines stored in lower-level caches. Some caches may be neither inclusive nor exclusive.

### 1.3.3. Classifying cache misses

As seen above there are several sources of cache misses. Not only can misses occur the first time a cache line is requested by a program, but there are quite a few reasons why such a line can be evicted from a cache after the first access and before a subsequent access. The misses can thus be classified according to the reason why they occurred:

A *cold miss* occurs when a line has never been requested before by the program. *These are the misses that would still occur with an infinite-size cache. Capacity misses* occur as a result of the limited size of the cache, while *Conflict* misses occur as a result of the limited associations and eviction policies of the cache. The rigorous definition uses a hypothetical fully-associative cache of the same size as the real cache using the ideal Bélády eviction policy [10]. This ideal eviction policy is to evict the line whose next access lies the furthest in the future. Misses that

would still occur on such a cache can be purely attributed to the cache capacity, while those that occur in the real cache but not this ideal fully associative cache are conflict misses, arising from the limitations of the cache management. The expression *working set* designates the collection of addresses used by a program in its operation. If the working set is larger than the cache capacity, capacity misses will occur. These three kinds of misses were defined in 1989 by Mark Hill [42]. Further development of the cache hierarchy added an extra source of miss, cache coherence. Such *Coherence* misses arise on memory shared between different threads. They occur when a line is evicted from a core as the result of another core requesting exclusive access to it (generally because that core is making a write to the line).

> **Takeaway:** Cache misses can be classified according to their cause: *Cold* or compulsory misses for lines never accessed before, *Conflict* misses for set related limitations. *Capacity* misses are caused by insufficient cache size, and *Coherence* misses occur when multiple threads access and modify a line.

In this section, we have thus seen the concept of management of individual caches, including sets and eviction policies, and the concept of cache hierarchy with its cache coherence and cache inclusivity aspects. We have also seen how this result in a classification of cache misses.

## 1.4. CPU and operating systems

General-purpose CPUs generally end up running several programs at the same time or being shared by several users. To permit such sharing and provide various abstractions that simplify programs, a special piece of code takes care of managing the system, the *Operating System (OS)*.

### 1.4.1. What is an Operating System

An OS is a piece of software whose purpose is to manage the hardware on behalf of user programs, and allow several programs to share it. To use the words from *Operating Systems: Principle & Practice* [3], an OS is a *referee*, an *illusionist* and a *glue*.

**Referee** : The OS ensures programs share the computer, playing nicely with one another, and decides on how to attribute resources and execution time. This also includes isolating programs from one another to limit what damage malicious programs can do.

**Illusionist** : The OS maintains the illusion to programs that each of them is alone in the system, has access to as much memory as it needs, and that many interactions with the outside world are easy. Typically user programs can just ask for a line input from the keyboard, without caring about all the steps involved, such as communicating over USB, waiting for the various key-presses, decoding them, and determining when a line has been written.

**Glue** : The OS provides integration between the different parts of the system, underpinning features like inter-application sharing of a clipboard, file system, and desktop environments.

OSes generally rely on features provided by the CPU to help with the various goals. Most notably, CPUs generally include at least 2 modes of operation, one of which has unfettered

access to the hardware, often known as kernel mode, and one that has much more limited capabilities, known as user mode. A major piece of the OS, known as the kernel runs in the aforementioned kernel mode, while the remaining programs will run in user mode.

The CPU may divert the flow of execution to specific routines executing in the kernel mode when Operating System help is needed to handle an event. This can be called *exceptional control flow* and can be divided into three different categories. Different ISAs use different terminologies, we will use Intel's.

An *interrupt* occurs when the outside world (peripherals) reaches out and the CPU needs to handle this request. These may occur at any point in time and are not related to the current instruction stream.

The other two diversions of control flow occur when an instruction requires the CPU to get help and are collectively known as *exceptions*. Depending on whether the control flow will return on the instruction that caused the diversion or the following one distinguishes *faults* from *traps*. The former re-executes the instruction and occurs usually when an instruction encounters an error condition (access right violation, division by zero, floating point operation when the corresponding hardware is disabled), and the instruction can be re-tried when the error is corrected. The latter, a trap, occurs when the exceptional flow replaces the instruction. The most common use of a trap instruction is when a program needs to reach out to the kernel for help when they need to take actions not possible in user mode. Such a request is often known as a *system call (or syscall)*. For example, to read or write from a file, a program must request the kernel who will then do whatever is necessary so that the file content appears in the memory of the program (this might even involve a user-mode process running the file system). The trap/fault distinction may be blurred in some ISAs, especially one when the instruction size is fixed.

Operating systems is a topic unto itself and we will focus on a few specific areas that are of relevance for this thesis. We encourage the curious reader to have a look at books such as *Operating System Concepts* [106] (known as the *Dinosaur book*), Operating Systems: Principle & Practice [3] and the freely available online *Operating Systems: Three Easy Pieces* [8], or for more introductory materials CS:APP [16].

> **Takeaway:** The operating system is in charge of managing the hardware on behalf of the user programs, acting as a referee, an illusionist, and glue in between the various programs. OSes rely on several CPU features, including CPU support for privileged (kernel) and unprivileged (user) modes of execution.

### 1.4.2. Scheduling

If a system runs several user programs, it must juggle several streams of instructions (at least one per program, but some programs can have several). We call *thread* such a stream of instruction. One key component of the operating system kernel is the *scheduler*, whose job is the share the CPU cores in between the various threads so that each one of them can make progress.

Most modern OSes use preemptive scheduling whereby the OS will, with the help of the hardware, interrupt threads at fixed intervals after having scheduled them and schedule the next thread. On multi-core systems, the scheduler is also in charge of deciding what thread

each core runs, and thus, once re-scheduled, threads may be executed on a different core than the one they were previously running before being interrupted.

To make this possible, the scheduler stores each thread's current execution state (its registers) when descheduling a thread, and restores those when the thread is scheduled.

> **Takeaway:** A thread is a given stream of instruction, and the scheduler is in charge of scheduling the threads using the various CPU cores on the system so that each thread can make progress, and of interrupting them to enforce the sharing. Through its execution threads may be scheduled on several cores.

### 1.4.3. Virtual memory and process isolation

In addition to threads, defined by being a stream of instruction and an execution state, a second important abstraction is that of the *process*. A process is a set of threads that will share most resources. The process abstraction is the granularity at which isolation is mainly implemented by operating systems. Many resources are protected behind syscalls where the operating system can obviously validate the requests, but memory cannot be hidden behind syscalls.

In order to abstract the details of the hardware memory (its size, the presence of areas reserved for hardware devices), hide the sharing of this memory among processes, and also to allow OSes to implement isolation and other techniques, a key feature provided by CPU is *virtual memory*.

Memory addresses manipulated by programs (or *virtual addresses*) are translated in hardware according to a mapping set-up by the OS to *physical addresses* composed of translation entries. As part of the process, the OS can also specify properties such as access rights, and possibly that an address is not mapped. The hardware reaches out to the operating system for help on memory accesses that are not valid per the translation structure (for instance an unmapped address or a permission violation).

To keep the data structure size manageable, this translation occurs at the granularity of *pages*, blocks of contiguous addresses of a given size, such as $4\,\mathrm{KiB}$, so that the translation applies to the higher bits of the memory addresses, while the lower bits remain unchanged. This data structure is usually laid out in memory, using physical addresses, and caches are used to speed up the translations. Many different structures have been proposed, and we refer the reader to Jacob and Mudge's *Virtual memory in contemporary microprocessors* [49] and Hennessy & Patterson [41] for surveys of translation structures.

The low bits of a virtual address are called *Virtual Page Offset (VPO)*, while the higher bits are called the *Virtual Page Number (VPN)*, similarly, the physical addresses are split in *Physical Page Number (PPN)* and *Offset (PPO)*. The MMU leaves the VPO unchanged (it is thus identical to the PPO), and determines the PPN from the VPN, or faults if the entry is not valid for the access purpose (insufficient permission, or unmapped address).

Most OSes include process isolation using virtual memory, granting each process its own address space, but it is also used for a variety of other purposes. Swapping is the sending of unused pages to the disk to make room in the physical memory for pages that are required. Various techniques involving sharing of pages by different processes to reduce the physical memory footprint may also be implemented such as Zero Free on Demand or Copy on Write pages. Those techniques usually involve setting permission bits to fault when a process makes

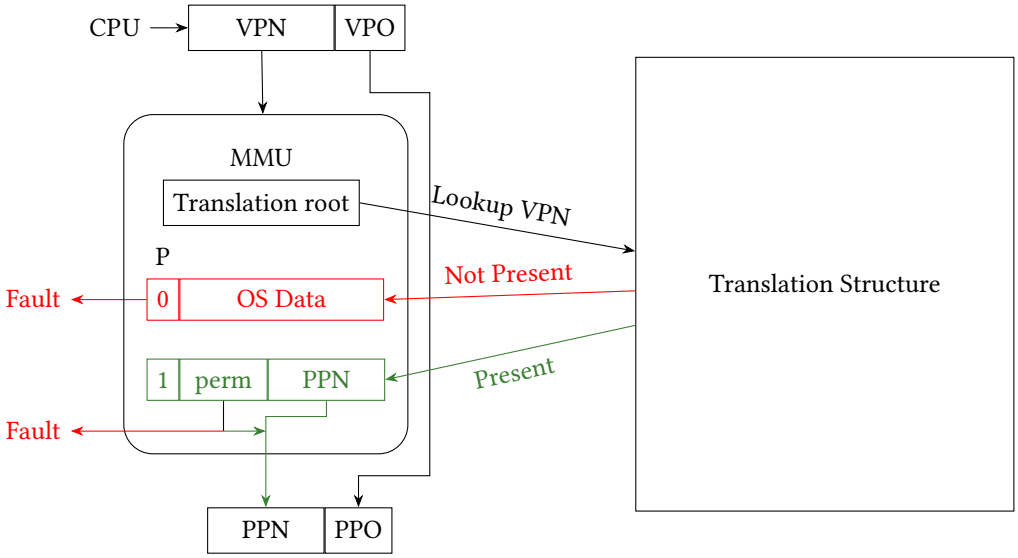



Figure 1.10.: Virtual to Physical address translation.
In this example P designates the Present bit, indicating if a page is mapped or not, and perm is translation entry metadata including permissions. A valid translation occurs if a present entry is found and it has appropriate permission, otherwise a fault will occur.

an access that is not compatible with the optimization and using the fault handler to take the appropriate action, such as granting a fresh page to the process.

On 64-bit CPUs, the virtual address space can be significantly larger than the physical address space. This permits a security measure known as *Address Space Layout Randomization (ASLR)* [103], where the location in the virtual address space of software, in particular, the kernel is changed each time the program is loaded in memory (at boot time for kernels). This measure hampers attacks that require knowing addresses in victim programs, by requiring them to find a way to extract this information before being able to run such an attack.

> **Takeaway:** Processes are sets of threads sharing resources. Virtual memory translates the *virtual addresses* manipulated by the processes into *physical addresses* and allows the OS to implement various techniques, including granting each process a separate set of mapping, maintaining process isolation, and the illusion of a simple and large memory space. It underpins Address Space Layout Randomization (ASLR), used to make attacks harder.

### Virtual Memory and Caches

Having introduced the distinction between virtual and physical addresses, what kind of addresses are manipulated within the cache system? Two subtle problems may arise when dealing with virtual addresses in the cache system [109]:

**Synonym addresses** : When a given physical address is mapped at two different virtual

addresses, it is usually specified in the architecture that writing to one should be observed from the other address[3].

**Homonym addresses** : When switching virtual address space, it is possible, and even likely, that a given virtual address will no longer refer to the same physical addresses.

As such, there are quite a few pitfalls in using virtual addresses within the cache system. In most systems, caches only deal with physical addresses. Sometimes, small caches may use the virtual address to identify the set, and then use the physical address to match the actual tag. This is still somewhat subtle to handle, except in the case when the set bits are part of the VPO/PPO, in which case using the virtual address to index is equivalent to using the physical one and faster.

This however leads to another issue. The virtual-to-physical address translation must be fast and requires memory accesses itself. To avoid a performance collapse, dedicated caches store the translation [16, 41, 109], the *Translation Look-aside Buffers*, or TLBs, with two-level TLBs becoming common in recent CPUs, and possibly extra caches [99]. Only when a translation entry is not found in the TLBs, must the MMU actually walk the translation structure, making memory accesses through the caches. This use of the cache is made possible by the physical indexing and tagging in those caches.

> **Takeaway:** To avoid the homonym and synonym problems, most caches are physically tagged. *Translation Look-aside Buffers (TLBs)* enable caches to deal with physical addresses by speeding up address translation in the common case.

In this section, we have thus seen a definition of the operating system and a few specific details of operating systems that matter to our research, including privilege modes, the scheduling of threads, the concept of process and that of virtual memory, and how it interacts with the cache system.

## 1.5. Intel CPU specificities

In this section, we will look in more detail at the specific family of CPUs manufactured by Intel, which were the object of this research. This will illustrate practically some of the concepts exposed previously. Some of the specificity is part of the ISA, also implemented by AMD CPUs. We refer to the ISA as x86 to distinguish it from the features specific to Intel's implementations.

### 1.5.1. Genealogy of relevant Intel CPUs

As shown in Fig. 1.11, Intel and x86 CPUs have a long history, which starts with the original 8086 in 1978. This is the first CPU in the x86 ISA, at the time 16-bit. Skipping a few iterations to 1985, the i386 CPU marks the move to a 32-bit ISA, and its successor, in 1989, the i486 introduces an L1 cache along with being the first microarchitecture using pipelining to increase performance. In 1993 the P5 microarchitecture introduces superscalar execution (several instructions are now

[3] In C, however, one must make aware the compiler of the possibility of such changes, otherwise this behavior is undefined and the compiler may optimize code in ways that will break this.

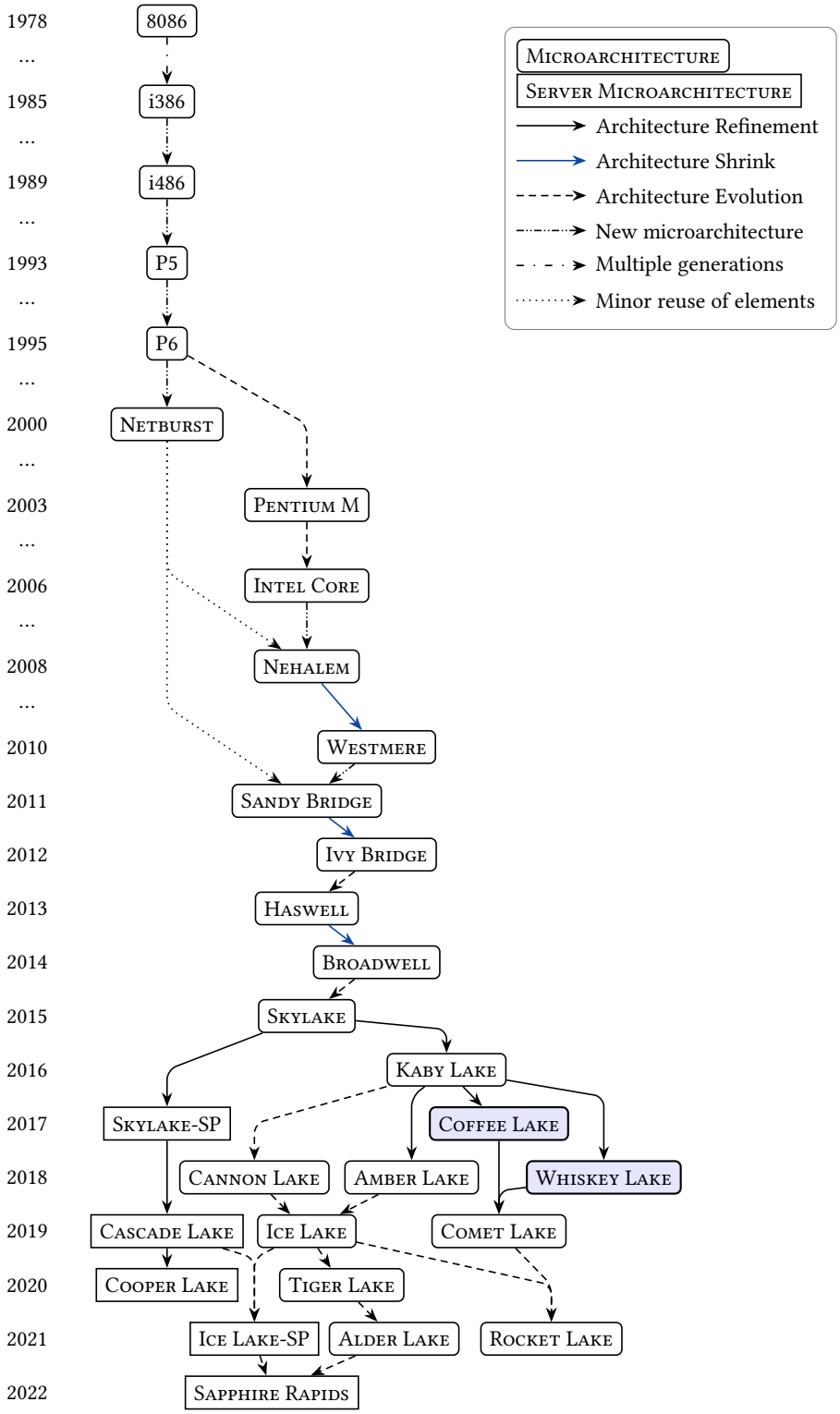



Figure 1.11.: Genalogy of Intel CPUs: Microarchitectures are dated using their introduction. Excludes the Atom lineage and is only exhaustive starting with Nehalem.

executed in parallel) and branch prediction. In 1995, the P6 architecture then features the first out-of-order execution design of the family.

Following the failure of the deeply pipelined Netburst architecture (2000), the Pentium M architecture (2003), derived from the P6 out-of-order design, becomes the starting point for the next stages of design evolution. At this point, the Netburst final CPUs (Prescott, 2004) are the first to feature the x86-64 ISA, the 64-bit extension of the Intel ISA designed by Intel's competitor AMD. Netburst also introduces 2-way SMT under the name *Hyper-Threading*. In 2006, the Intel Core microarchitecture marks the move to 64-bit of the P6 and Pentium M line, it nevertheless retains their lack of hyperthreading.

It is followed by Nehalem (2008), which starts the consolidation of Core and Netburst, and its die shrink Westmere (2010), a minor evolution built using smaller transistors. It is a superscalar out-of-order design with hyper-threading, characteristics that will be retained in all later architectures. This is the start of a period where Intel alternatively designs a new architecture on the current process and then shrinks it to a new process with few changes. In 2008, Intel also starts a line of low-power CPU designs, named Atom, which are unimportant for this thesis. Those were much simpler designs, with no super-scalar out-of-order execution until very recently, but also had regular microarchitecture evolution.

In 2011, the Sandy bridge architecture introduced a cache architecture that has been conserved for over a decade, which will discuss later on (Section 1.5.2). It was shrunk into Ivy Bridge (2012). Haswell (2013) was then a new architecture, which was shrunk into Broadwell (2014), and then the Skylake architecture was introduced in 2015. At that point, however, Intel's manufacturing process stalled, at the node identified as 14 nm, and the following years saw several revisions of the overall Skylake design, which we will discuss later.

In 2017, the Skylake server design named Skylake-SP featured a new cache hierarchy, with an L3 that is no longer inclusive and a new interconnect topology, a *mesh* while using a core derived from the Skylake CPU core. There is now a lineage of server CPUs coexisting alongside the lineage of client CPUs, where core designs are the same, but caches and interconnects differ.

The client design iteration started with Kaby Lake in 2016, as an optimization of Skylake. In 2017, Coffee lake further refined the design for workstations. In 2018 two other refined designs derived from Kaby Lake for lower power usage were introduced, the Whiskey Lake and the Amber Lake designs. The Cannon Lake architecture, on the next process node, featured a single released CPU in 2018, owing to manufacturing defects.

In 2019, the Ice Lake CPU was the first microarchitecture manufactured at scale on the new 10 nm process and targeted the lower power end of the client CPUs with a new core design (Sunny Cove).However, it still retains Sandy Bridge's interconnect topology. Meanwhile, the high-power end saw the Comet Lake design on the old 14 nm process, the final iteration of the Skylake client design. The Tiger Lake CPUs were then released in 2020, using the Willow cove core design, optimized for the 10nm process.

On the server side, Cascade Lake appeared in 2019, an optimized Skylake-SP using the same 14 nm process. In 2020 the Cooper Lake derivative, again using the 14 nm process, targets the multi-socket systems, while the 10 nm Ice Lake-SP was released in 2021, using the same Sunny Cove core design as Ice Lake but the mesh interconnect of the server lineage.

In 2021 Rocket Lake is a backport to the 14nm process of the 10 nm Willow Cove core from Tiger Lake targeting the higher-power market in replacement of Comet Lake, finally ending

the Skylake family of architectures. Intel also introduces its first mainstream CPU to include two different cores, Alder Lake. This *heterogeneous* design includes large Performance (P) cores, based on Golden Cove (the successor to Willow Cove), and Efficiency (E) cores, derived from the core used in Atom CPUs. Alder lake still appears to use a ring topology.

The Golden Cove core is also used on the 2022 Sapphire Rapids server CPUs, where those cores are connected with a mesh topology.

> **Takeaway:** This research used a Coffee Lake and a Whiskey Lake CPU and should apply to CPUs from Sandy Bridge to Comet Lake.

### 1.5.2. The Sandy Bridge lineage's cache structure

Starting with the Sandy Bridge Architecture, Intel introduced a cache structure that was conserved over many architectures, including in all client Skylake derivatives, with only minor changes in the cache sizes.

Figure 1.12 shows the general structure used by this cache. At the first level, on each core, the instruction and data memory access paths each hit their own small caches in 4-5 cycles (L1D and L1I). At the second level, each core has an L2 cache, that serves the L1 misses in 15-20 cycles. At the last level, a shared L3 acts as the last-level cache and answers in 50-100 cycles, while memory takes over 200 cycles.

#### The Sandy Bridge Uncore and last-level cache

**The last-level cache is divided into slices for performance reasons.** The bigger SRAM is, the slower it is to access, and in addition, more cores mean higher request traffic to the cache hierarchy. To make the last-level cache scale properly with the increasing number of cores, the cache is split into several slices, each paired with a core. Each slice is a cache but they collectively work together to act as a single large shared cache. Each physical address is mapped by a hash function to a single slice. The hash function used is a *complex hash function* that is not documented further by Intel. We will see in Chapter 3 how this hashing function has been reverse engineered, and Appendix A will provide additional details.

Chips with more cores have proportionally more slices, which can proportionally serve a greater number of requests. Consequently, this designs allows the L3 to scale with the number of cores.

**The last-level cache is distributed over the *Uncore*.** Modern CPUs tend to have several distinct clock domains. Each core can vary its frequency independently, but a significant part of the system is not part of a core. Consequently, a common clock domain is needed for the interconnection network in between the cores, the GPU, the memory, and I/O systems. This part of the core, *i.e.*, everything that is not a specific core, is called the *uncore*.

Prominent in the uncore is the core interconnect, which is not well documented by Intel apart from stating it is a bidirectional ring ([46], Section 2.4.5.3). This leaves room for several interpretations and topologies.

Table 1.1.: Specific cache parameters for various Intel CPUs

| CPU | i7-2630QM | Core i5-8365U | Core i9-9900 |
|---|---|---|---|
| Year | 2011 | 2019 | 2019 |
| Microarchitecture | Sandy Bridge | Whiskey Lake | Coffee Lake |
| Stable Frequency | 2.00 GHz | 1.60 GHz | 3.10 GHz |
| Max Frequency | 2.90 GHz | 4.10 GHz | 5.00 GHz |
| L1I size | 32 KiB per core | 32 KiB per core | 32 KiB per core |
| L1D size | 32 KiB per core | 32 KiB per core | 32 KiB per core |
| L1D latency (cycle) | 4-5 | 4-5 | 4-5 |
| L2 size | 256 KiB per core | 256 KiB per core | 256 KiB per core |
| L2 latency (cycle) | 11 | 12 | 12 |
| L3 size | 8 MiB ($4 \times 2$ MiB) | 6 MiB ($4 \times 1.5$ MiB) | 16 MiB ($8 \times 2$ MiB) |
| L3 latency (cycle) | 26-31 | $> 42$ | $> 42$ |
| Max DRAM | 32 GiB | 32 GiB | 128 GiB |
| DRAM latency | 28 cycles + 49 to 56 ns | 42 cycles + 51 ns | 42 cycles + 51 ns |

The interconnect network connect several nodes, which each contain a physical core and a last-level cache slice. It also includes a node for the *system agent*, which manages communication with the off-chip world, including the DRAM main memory; and a node for the Graphics Processing Unit (GPU) when the chip contains one. While it was usually assumed that each core had exactly one slice, it is no longer the case on some recent Intel systems [124]; starting in Skylake, it appears that each cache core may contain two cache slices.

Figure 1.13 is a die shot, annotated by WikiChip [21] of the 8-core Coffee Lake CPUs. This layout is used to produce, among others, the Intel Core i9-9900 CPU, one of the CPUs used in this research. The die presents the cache structure described in Fig. 1.12, with 2 visible slices per core, a particularity present in the 14 nm Skylake architecture and its client derivatives.

### Cache Coherence

Intel discloses it uses a cache coherence protocol named MESIF [46, 74, 75]. It is an extension of the MESI protocol described earlier, with an additional F state used to optimize data sharing between sockets. This fifth state, Forward, designates one CPU responsible for answering requests to share the data [41]. There can be at most one CPU in the F state alongside zero or more CPUs in the S state.

### Multi-socket system

A multi-socket system is a system where several multi-core CPUs, each with its cache system, share a single physical memory space with an interconnect between the two packages. In

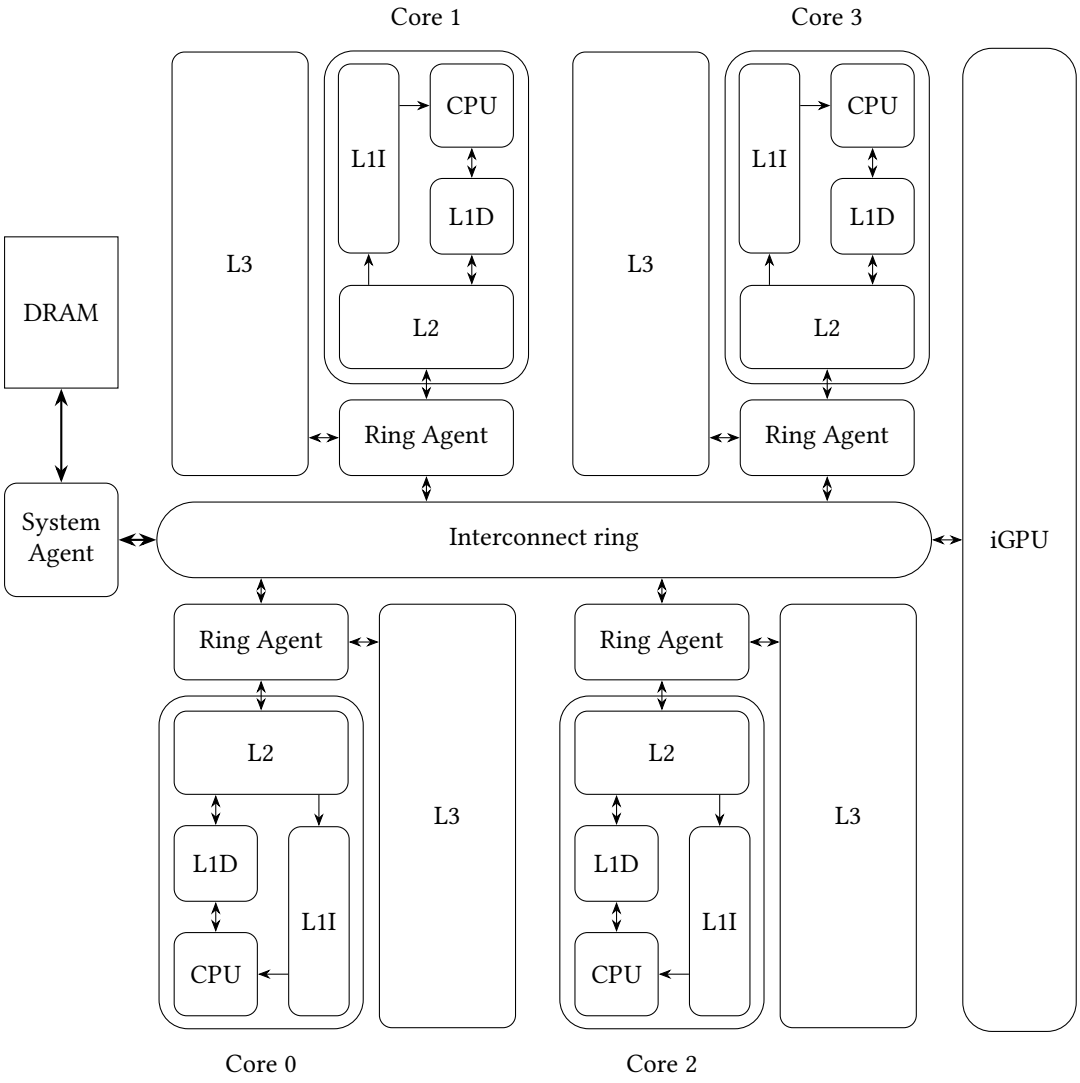



Figure 1.12.: Sandy Bridge-like cache structure, for a 4-core CPU.
Here CPU designates what is also called the Processing Element (PE) in ARM documentation, that is the fetch and out-of-order logic excluding the memory system.

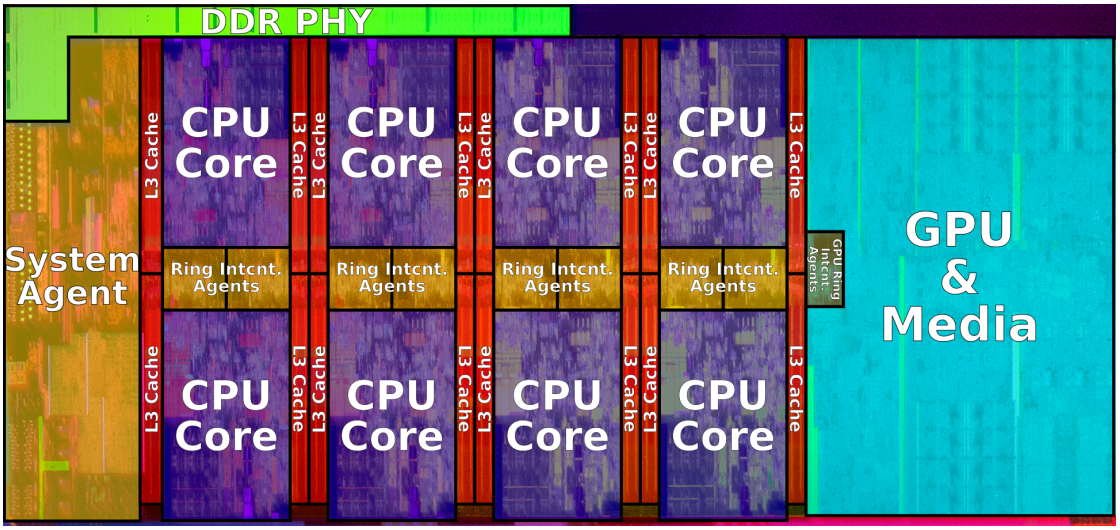



Figure 1.13.: Intel Coffee Lake 8-core die shot.
Image by Intel, annotated by WikiChip [21].

multi-socket systems, there is no single last-level cache ensuring the coherence between the caches of the two cores.

To maintain coherence, it is usually necessary to send *snoop requests* over the interconnect between the sockets. However, in some systems, it seems that some of the Error Correction Code (ECC) bits inside the DRAM are used to maintain some coherency metadata [46, 75]. Regardless, requests may need to flow in between the two sockets as each socket acts as the controller for part of the main memory.

> **Takeaway:** The Sandy Bridge cache structure, used by the Coffee Lake and Whiskey Lake CPUs we studied, is a 3-level hierarchy with an inclusive L3, distributed in slices, divided among the various cores. It uses the MESIF cache coherence protocol.

### 1.5.3. Subsequent evolution of Intel caches

By the time the Broadwell EP and EX CPUs were released, with up to 24 cores connected by a pair of interconnect rings, it was becoming clear this ring structure would not scale to higher core counts.

Because of this, a mesh interconnect was adopted for the server-class Skylake CPUs with a higher number of cores [6]. This interconnect seemed to have been originally developed as part of the Xeon Phi platform [111]. With an increasing number of cores, a shared, inclusive last-level cache would be under increasing pressure and be at risk of causing evictions in the lower level caches owing to conflict in its cache sets of the working sets of each core. In addition, this inclusivity wastefully duplicates the entire L1 and L2 in the L3 cache. Consequently, at the same time as the mesh topology was included, Intel switched to a non-inclusive L3 cache, complemented with cache directories to maintain cache coherence. The L1 and L2 sizes were

then also adjusted to take advantage of the size constrained imposed by inclusivity.

Because of this divergence in the cache hierarchy, the client (Skylake) and server (Skylake-SP) CPUs are deemed to be two different architectures, sharing a very similar core design. The core designs are not identical, however, owing to the addition of a few ISA extensions, such as AVX-512, to the core used in Skylake-SP.

Subsequently, the server CPUs have all retained this mesh structure, while the client CPUs retain, as of Alder Lake, a layout using an interconnect ring. However, the Tiger Lake CPU also changed the L3 to be non-inclusive [119].

### 1.5.4. Caches and the x86 ISA

In theory, caches are supposed to be part of the microarchitecture and should not be visible from the ISA and can only be observed through timing and other side channels. However, in practice, this is not the case.

The split between instruction and data caches implies specific rules must be followed for self-modifying code, arising from those caches. There is information available about those caches through the `cpuid` instruction and MSRs, and the OS may even disable caching entirely. The OS must also care about memory range types, for some physical memory is used by devices and must not be cached for correctness.

The x86 ISA also provides a few instructions that manipulate the cache state.

#### `clflush`

The *Intel 64 and IA-32 Architectures Software Developer's Manual* [47] documents the `clflush` instruction as follow :

> Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

It is one of the few areas where caches do appear at the ISA level. This instruction's purpose is generally to deal with specific devices, where ensuring the write-back is necessary for correctness, or to manually manage caches for performance purposes, for instance, when the eviction policy would pick the wrong victim.

However, security researchers have hijacked this instruction to build various attacks, as described in Chapter 3. Notably, the coherence domain corresponds to all caches on all cores in all sockets.

The `clflushopt` instruction is a similar instruction with relaxed ordering properties. Incidentally, these relaxed constrained have allowed the use of this instruction to improve Row Hammer attacks [22].

> **Takeaway:** The `clflush` instruction can be used to evict a line from the entire cache coherence domain.

#### The software prefetch family of instruction

The x86 ISA also contains several instructions informing the CPU that an address will be needed in the near future and should be brought into the cache when possible. These are the `prefetchH` family of instructions and the `prefetchw` instruction, documented in Volume 2 of *Intel 64 and IA-32 Architectures Software Developer's Manual* [47]. These instructions are treated as hints and may be ignored by the CPU. They can be used to reduce the number of cold misses encountered by a program.

The next chapter discusses prefetching in more detail. Nevertheless, it should be noted that getting a performance boost out of these instructions requires invasive changes to the source code. Furthermore, those changes are dependent on a given microarchitecture, and thus reduce the portability of the program. Consequently, software prefetch is rarely used in portable programs.

> **Takeaway:** The x86 ISA contains software prefetch instructions, but those are generally not used in portable programs.

### 1.5.5. x86 Virtual Memory

We have seen in Section 1.4.3 the general concept of virtual memory. However, as various architectures have used varied translation structures, we have deferred this discussion until this section, in which we explain the structure used on Intel CPUs.

x86 CPUs have had a long history, with the virtual addresses used moving from 16 to 32 and then 64 bits, and thus support several paging modes (with different address sizes), in addition to a feature called *segmentation*. This research has only studied 64-bit CPUs running in 64-bit mode (aka *long mode* in Intel's documentation), so we will only concern ourselves with the details of the paging scheme used in this mode. In long mode, the segmentation feature has been pared down significantly and can thus be ignored.

x86 CPUs use a page size of 4 KiB, that is, 4096 bytes; such pages are aligned on 4 KiB boundaries. The general translation mechanism used is *multi-level page tables*. The Virtual Page number is split into several level indexes, with the most significant bits being used first.

Each level uses its slice of bits to index into a table, which gives the logical address of the table used to translate the next level until all the VPN bits have been used, as shown in Fig. 1.14. It is possible to mark an entry as non-existent and stop the entire translation early. This leads to a sparse and efficient structure. A privileged register, CR3, is used to indicate the physical address of the first level page table.

One additional feature is the ability to stop the translation one or two levels early and create *huge* pages (2 MiB or 1 GiB for 64-bit paging), where the last or last two indexes become part of an extended page offset inside the much larger page. (Such pages are aligned to their respective size).

The general principle above apply more generally to other paging mode supported by Intel CPUs. For the 64-bit mode, those CPUs use entries of 64 bits in both the last-level page tables and the higher-level *page table directories*. Each table (or directory) is similarly 4 KiB in size, aligned on 4 KiB boundaries; this results in each level being able to fit 512 such entries, consuming 9 bits of the address. Each entry contains the next-level structure's PPN and metadata bits.

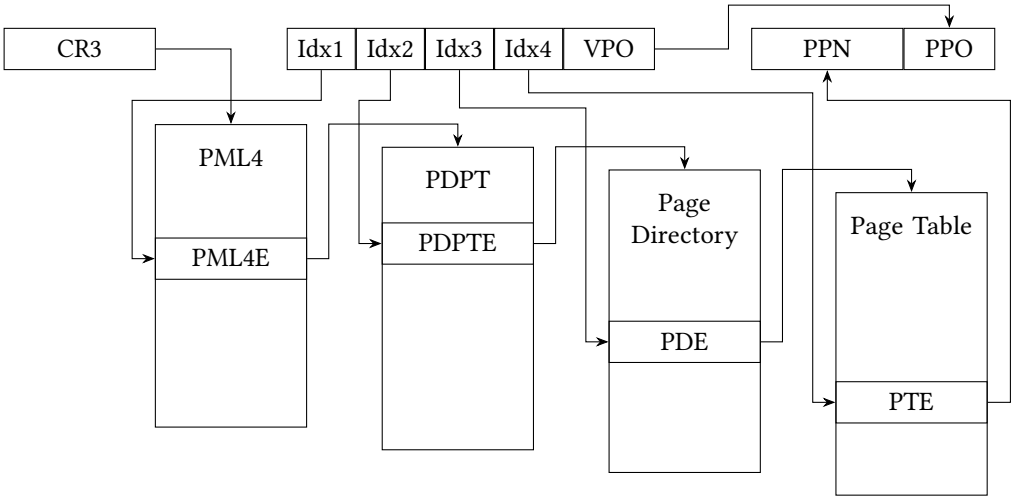



Figure 1.14.: Four level paging in 64-bit mode on x86 CPUs

Currently, CPUs support either 48 bits or 57 bits virtual addresses (where the most significant bits must be sign-extended from bit 47 or 56). This corresponds to 4 or 5 levels of page tables. Physical addresses are limited to 52 bits according to the ISA Manual as of July 2022.

Our systems used 4-level paging, which we describe in Fig. 1.14. Intel has a specific name for each level of the page table and for the entry they contain. We list them here in reverse order of traversal, which is the order in which they were introduced as Intel slowly increased the number of supported levels.

**Page Table (PT):** The last level of translation for normal-sized pages. It contains Page Table Entries (PTEs).

**Page Directory (PD):** The second to last level. It contains Page Directory Entries (PDE) that provide the physical address of a Page Table or of a 2 MiB page.

**Page Directory Pointer Table (PDPT):** Third to last level (second level in 4-level paging), it contains Page Directory Pointers Table Entries (PDPTE), that point to a Page Directory. It may also point to a 1 GiB page.

**Page Map Level 4 (PML4):** This is the first level in 4-level paging mode, it contains PML4 Entries (PML4E), that provide the address of a Page Directory Pointer Table.

**Page map level 5 (PML5):** In 5-level paging mode, this corresponds to the first level, it contains PML5 Entries (PML5E), that provide the physical address of a PML4.

> **Takeaway:** Intel CPUs use a multi-level page table structure for their virtual memory, with 4 KiB pages, and 2 MiB or 1 GiB huge pages.

## Summary

In this chapter, we have defined computers, their architectures, and their microarchitectures; discussed modern CPU design constraints and how fast CPUs are made; discussed the operating system and key features that matter for our research; delved into modern cache organization, and examined Intel CPUs specifically, particularly their caches.

# 2. Hardware prefetching

As we have seen previously, the memory wall is one of the main impediments to CPU performance. Therefore, reducing the number of caches misses in program execution is one of the main ways to improve program performance. Various techniques to reduce capacity, conflict, and coherence misses exist, but *cold* misses, also known as *compulsory* misses, are harder to deal with. To avoid such a miss, one would need to fetch into the cache data before it is needed by a program.

This can be done in two ways :

**Software prefetching:** In some cases, user programs can anticipate what data they need and could inform the hardware of it before they need it. Thus, software prefetch instructions have been introduced in ISAs such as x86, as shown in Section 1.5.4. However getting a performance improvement by using them usually requires optimizing for a specific microarchitecture, and complex code changes. This technique is thus rarely used.

**Hardware prefetching:** Some patterns of memory accesses can be quite predictable, and consequently, one can attempt to build hardware predictors that, as branch predictors do, make accurate guesses of addresses that are likely to be used in the future.

Manufacturers have deployed hardware prefetchers in modern CPUs to gain competitive advantages. However, because they belong solely to the microarchitecture, disclosures about these are limited, and the documentation is very sparse, which motivates our study of prefetchers.

In this chapter, we review the field of prefetcher design, starting with the general principles (Section 2.1) and an overview of the various classes of prefetcher designs in Section 2.2. Then we will restrict ourselves to the specific class that is the subject of this research, Stream prefetcher, looking first at the designs proposed in academic literature (Section 2.3) and then at the industry disclosures (Section 2.4). Finally, reverse engineering of the prefetcher will be covered as part of Chapter 3.

## 2.1. General concepts

A *Hardware Prefetcher* predicts which cache lines are likely to be requested by a CPU by observing patterns of memory accesses; and fetches those lines into a cache. Successful prefetches will turn a miss into a hit, and prefetching is the only way the hardware can turn a cold miss into a hit. As it is done in hardware, it does not require changes to users' programs, unlike software prefetching. There are three challenges for prefetchers:

**Prediction:** The first and most obvious challenge is a prefetcher that makes correct guesses. On the one hand, should a prefetcher make too many predictions to unneeded addresses, it

may cause contention, both with legitimate memory traffic and for space in the caches and buffers with valuable data. On the other hand, a prefetcher is only beneficial if it makes predictions to addresses that are used later. Making no prediction has no performance benefits. Two metrics are used to judge the precision of the predictions: *Accuracy* and *Coverage*. *Coverage* designates the fraction of CPU requests fulfilled by a prefetch. In other words, coverage is the fraction of misses avoided by the prefetcher. *Accuracy* designates the fraction of the prefetches that were used. It is generally easy to increase coverage at the cost of accuracy. However, this increases the pressure on the memory hierarchy and quickly hits the point of diminishing returns [32].

**Timeliness:** A second challenge is that of timeliness. While we have not investigated timing much in this work and will primarily focus on the predictions, prefetchers are only worthwhile when the fetches occur at the right time. A prediction made after the user request was already issued or too shortly before does not improve performance and only increases the load on the cache. On the other hand, a prediction done significantly too early may pollute the cache and evict data that is still needed, or the prefetched data may get evicted before it is needed. Consequently, prefetchers must consider the timing of memory access and carefully time the prefetches. It is worth noting that this timeliness issue is one of the main challenges that hamper software prefetch, as where prefetches must be inserted in the instruction stream to be timely depends on the microarchitecture. In addition, determining when to issue those prefetches remains challenging even with a fixed microarchitecture.

**Placement:** The last challenge is determining where to place the prefetched data in the memory system. Possible destinations include the various cache levels and CPU registers, but dedicated buffers could also be added and used. Indeed, some of the first prefetch papers used dedicated buffers to store prefetched data [51, 52, 88].

It is also important to remember that hardware prefetchers are subject to the same delay, die surface, and power constraints as other CPU components. Therefore, an adequate performance increase must offset the increase in surface and power, and the algorithm implementation must meet timing constraints. These constraints may limit the size of the data structure used and the algorithm's complexity, especially when it comes to the lowest level of caches, such as L1.

The primary input these components have is generally the sequence of requests made to the level of the memory hierarchy on which they operate. For instance, a prefetcher operating at the level of an L1 cache will observe all the requests from the CPU to this cache, possibly including both virtual and corresponding physical addresses. On the other hand, a prefetcher operating at the L2 cache would only see the sequence of misses from L1, which contains much less information and only includes physical address information.

Prefetchers are entirely part of the microarchitecture and have no existence in the ISA. In theory, their only impact is performance related, and they are part of the fierce competition among manufacturers for the CPU performance crown. Consequently, manufacturers are loath to disclose details about these prefetchers. While manufacturer optimization guides (e.g., the Intel 64 and IA-32 Architectures Optimization Reference Manual [46]) inform programmers of patterns to adopt to improve performance, disclosures from the industry on the actual behavior are otherwise very sparse.

**Takeaway:** *Hardware prefetchers* reduce misses by fetching into the cache lines before the CPU requests them. A good prefetcher must solve three challenges:

- precision, with good *coverage* of CPU requests and *accuracy* at guessing lines that will be needed
- *timeliness*, neither too early nor too late
- *placement* of the prefetched data

It must solve these challenges under the constraints of both the silicon budget and the information available where it sits in the cache hierarchy. Prefetchers are deployed in real CPUs but are sparsely documented.

## 2.2. Classes of prefetchers

Similarly to how branch prediction is a vast field (see Section 1.2.4), there are many different prefetcher designs. This section will overview the different classes of prefetchers that have been designed.

Different types of caches observe different types of access patterns, and this is especially true of the distinction between the instruction and data caches. Consequently, we will generally distinguish instruction prefetchers from data prefetchers. Our research has focused chiefly on the latter, so we will keep our coverage of the former concise. The cache level also matters, as prefetchers operating on the higher levels only see the subset of the memory requests from the CPU that have missed in lower level caches. Depending on the memory hierarchy designs, prefetchers may also observe virtual addresses or only physical addresses. Often, prefetchers do not have a connection to the MMU and deal in physical addresses.

We present here an overview of the classification, and we refer the reader to *A Primer on Hardware Prefetching* [32] and other recent surveys of the field [73] for more details.

### 2.2.1. Instruction prefetching

Due to the specific access pattern of instruction fetches, specific classes of prefetchers apply. We will see several approaches that have been used to prefetch instructions.

As instructions are laid out sequentially, the simplest prefetcher is a *Next Line Prefetcher*, which fetches the line following the load. This strategy works for linear code sequences but will obviously be wrong upon encountering branch instructions. As a result, given the frequency of branches, this approach leads to a prefetch accuracy of around 50% [32].

Recall from Section 1.2.4 that modern CPUs rely on branch prediction to feed the CPU pipeline. The logic evolution is thus to provide the prefetcher with information from the branch predictor in the CPU Fetch stage, leading to *Fetch Directed Prefetchers*. Consequently, these prefetchers see an accuracy that improves to over 50%. One addition to this technique is to prefetch both sides of a conditional branch, a technique called *Wrong Path Prefetching*.

It is worth noting that if a loop body fits in the cache, a prefetcher need not fetch it repeatedly, unlike the fetch stage, which may need to fetch the loop body for each loop iteration from the

closest cache where the loop fits. An instruction prefetcher generally has to concern itself with discontinuities in the stream of instructions. A *discontinuity predictor* predicts discontinuities in the instruction stream and uses this to guide the prefetch. This requires a few KiB of memory to implement but improves the prefetcher accuracy, exceeding 50%.

Another approach with a similar cost is to have a helper thread run ahead of the execution stream, identifying and executing only the instructions that influence control flow, a technique known as *prescient fetch*.

We will now look at approaches that significantly improve the accuracy at the cost of larger memory storage: *Temporal Streaming* consists in memorizing sequences of L1 misses and replaying them when encountering a miss in the sequence. As returns from functions and procedures are hard to predict, it is also possible to make use of the Return Address Stack to identify and disambiguate locations for calls and returns. This is called *Return Address Stack directed prefetching*.

One last approach, *Proactive prefetching*, uses the L1 references (and not just misses) from instructions that retired to learn patterns and then replay those patterns. It also records, separately, execution caused by kernel execution. Although such an algorithm has a memory cost of hundreds of KiB, it can reach an accuracy above 99%.

> **Takeaway:** A variety of approaches are used for instruction prefetching, with varying levels of accuracy and hardware cost. The accuracy easily reaches 50%, and more advanced methods can reach 99% accuracy.

### 2.2.2. Data prefetching

Data prefetching and prefetching in unified caches have different characteristics compared to the prefetching of instruction above. As this thesis will look at data prefetchers, we will spend more time discussing this side of prefetching and the classes most relevant to this work.

#### Next-line, Stride and Streams

This first class of prefetchers derives from the next-line approach used for instructions. Many CPUs since the 1970s have included such prefetchers. *Stream* prefetchers are designed to fetch sequences of consecutive cache lines in ascending or descending order. *Stride* prefetchers extend this to fetching sequences of lines separated by a regular stride, e.g., every third address. To do so, they usually have to identify the stride exhibited by the memory access pattern.

In both cases, such prefetchers have to deal with the fact that CPUs may do additional accesses in between accesses that are part of the stream of stride pattern. These patterns may also be jumbled by out-of-order execution. In addition, not all workloads exhibit suitable patterns. For example, array and matrix algorithms often exhibit clean striding or streaming patterns, but pointer-based structures such as linked lists or trees usually do not. (Sometimes, the memory allocator lays out such structures that the access pattern does end up with some regularity, but this is often not the case). Such prefetchers generally have two important parameters :

**Prefetch Degree:** The prefetch degree indicates how many distinct lines are fetched upon a single access that is part of the pattern.

**Prefetch Distance:** The prefetch distance indicates how far from a given access the prefetch may fetch lines.

These prefetchers generally need a structure to store identified streams, along with those in the process of being identified. The entries in those tables are sometimes tagged with information like the Program Counter (PC) of the instruction responsible for a pattern of access. (This is a trade-off as it may hide some patterns of access involving different instructions, but may avoid seeing a pattern where there is none or prefetching on an access by a different instruction that is mistaken as belonging to a pattern).

> **Takeaway:** Stream and Stride are a natural evolution of the Next-Line prefetcher for instructions. Their effectiveness is very situational, but they are simple to implement and are known to have been deployed in various CPUs.

### Temporal address correlation

This class of prefetchers seeks to exploit repeated traversal of the same data structure. They arise from the observation that stream and stride prefetchers are pretty inefficient when dealing with pointer chasing structures but that many workloads repeatedly traverse large pointer chasing structures.

Consequently, this kind of prefetcher records sequences of misses and attempts to replay them when the same sequence seems to restart. In a way, they are similar to instruction temporal streaming. They can also be understood as finding correlations between accessed addresses. , However, this represents an extensive state to manage, and diverse approaches have been used to attempt this [32].

Simple approaches can reach an accuracy of 30% on general workloads, and more refined approaches can average 50% accuracy for a few dozen KiB of storage. However, these prefetchers cannot guess addresses that have never been accessed before, unlike Stream and Stride prefetchers.

> **Takeaway:** Temporal address correlation attempts to identify sequences of addresses accessed together to replay them appropriately. They are effective for workloads with large-scale data structures that are repeatedly accessed.

### Spatial correlation

This class of prefetchers tries to exploit similar spatial patterns, generally resulting in regular but not necessarily striding data structures. In a way, this is a generalization of striding access patterns.

Generally, the challenge here is twofold: on a given access, one must first look up the correct structure of the access pattern and then identify a way to compute the other accesses in the structure. Challengingly, the address of the memory access itself cannot be used as the key.

One approach is to monitor the *deltas* between two accesses, that is, the relative distance between them instead of the absolute addresses. In some cases, one can identify repeated

patterns in the sequence of deltas caused by the repeated instantiation of a given structure in memory. Deltas between the first two addresses can then be used as a key.

More refined approaches may also incorporate the PC of memory accesses. Many different data structures have been proposed, and some prefetchers can achieve coverage of 95% with an 80% accuracy with less than a KiB of storage [32].

> **Takeaway:** Spatial correlation attempts to identify relative structure between accesses and can speed up accesses when a structure is instantiated many times in memory. These prefetchers may reach pretty good coverage and accuracy.

### Execution-based

These prefetchers do not rely on any regularity but simply attempt to run ahead in the execution sequence.

These techniques generally have to use some execution resources, such as execution units, to be able to run ahead. They, however, have access to many details that the other techniques do not have, being able to look at the actual instruction stream. This prefetcher has the drawback of requiring deep integration into the core and cannot be added on the side of an existing core. It also has to process what is, in a way, far more data than what prefetchers that only observe the actual memory accesses see.

These can be implemented in different ways. One can attempt to identify the loads and their dependency graph, use a helper thread or helper core running in parallel or use the core execution resources while the core is stalled[1].

> **Takeaway:** Execution-based prefetching attempts to look ahead in the instruction stream to identify loads earlier and prefetch the referenced lines in the caches before the core execution reaches the load.

### Adaptative prefetching

Orthogonally to the previous four classes of prefetchers, it is possible to include a feedback system that can tweak prefetch parameters depending on variables such as the prefetcher's success rate and the current pressure on the memory system or, for instance, arbitrate among several prefetchers in the system. Tweaks can also be exposed to software through model-specific registers.

This can adjust the prefetch behavior to fit the workload better. One interesting example is proposed by Pakalapati et al. [87], in which several simple L1 prefetchers are implemented and Instruction Pointers are classified so as to pick the most adapted prefetch strategy for each of them.

> **Takeaway:** In addition to the various techniques above, adaptative prefetch can make the CPU prefetch fit better a given workload.

Effective prefetchers are sometimes surprisingly elegant designs extracting deceptively simple patterns. The Best-Offset Prefetcher is an example of such a simple concept [72]. It fetches

[1] This approach has been implemented and disclosed by IBM, which we discussed in Section 2.4.2

lines at an offset from the memory accesses while continually monitoring memory accesses to determine the best offset to use. It does so by matching the latest access with recent accesses to determine if a given offset would have prefetched it.

#### Summary

Many approaches have also been developed for data prefetching, with an even richer literature. However, it is worth noting that unlike the 99% accuracy reached in instruction prefetching, data prefetchers generally have a much lower accuracy outside of specific patterns.

The prefetcher we will later investigate belongs mostly to the Stream prefetcher category, so we will focus on this class of prefetchers in the remainder of the chapter.

## 2.3. Stream prefetch in academic publications

After having a general overview of prefetcher concepts, we will now look in more detail at various designs of stream prefetchers proposed in academic papers. The Stream prefetcher was first proposed in a paper by N. Jouppi [51], along with a Technical Note [52]. Recall that a *stream* is a sequence of accesses to consecutive lines in increasing or decreasing order.

This first incarnation of a Stream prefetcher uses a FIFO buffer to store the outgoing prefetch requests and, after the requests have been completed, the prefetched value. On an initial miss, the buffer is filled with the successors of the requested block, and prefetch requests are sent. On a cache look-up, the stream buffer head is searched in parallel to the cache. The buffer is dimensioned to hide the read latency. This buffer is flushed when a second miss occurs. This flush is obviously inefficient if, say, two streams are traversed in parallel by the program (for instance, while executing a long string compare (`strcmp`). This prefetcher is pretty efficient for instruction fetches that generally contain a single stream, unlike data accesses. An improved design is then proposed in the paper to improve data prefetching.

The multi-way stream buffer generalizes the previous design with several buffers in parallel. When a miss occurs, an LRU policy is used to pick a buffer to flush a buffer and instantiate a new stream.

One addition that is only present in Jouppi's technical note [52] is the concept of *quasi-sequential streams*. This technique adds several comparators to the stream buffer, which permits skipping addresses in the stream. This concept of imperfect streams is acutely relevant today, with out-of-order execution scrambling the observed streams.

Placharla et al. [88] pointed out that only initializing a stream for memory accesses that appeared to be part of such a pattern, instead of initializing on any miss, improved accuracy with a minimal decrease in coverage. Generally, they suggest a stream be allocated when a miss occurs to line $i + 1$ shortly after a miss to $i$.

Sherwood et al. [104] introduced the concept of confidence in stream prefetcher to determine whether to allocate a stream. It is worth noting this confidence is updated using actual accesses from the program. They also add a concept of priority to decide among several streams whose prediction to prefetch.

Hur and Lin [44] proposed another adaptative stream prefetcher, *Adaptative Stream Detection*. It builds a *Stream Length Histogram* and uses it to optimize how far to prefetch.

Lastly, Srinath et al. [112] detail an adaptative prefetch with the following behaviors: It allocates an entry upon a first miss if no matching entry is found. Then, it uses the subsequent two accesses in a $\pm 16$ line window around this line to train the prefetcher direction. If those two accesses match the direction, the stream switches to the *Monitor and Request* state. In this state, the prefetcher monitors memory accesses within the window associated with the stream. When access lies within the window, a set number, $N$, of lines are fetched beyond the end of the window. It will then shift its window by $N$. The prefetcher then tracks some metadata to monitor its effectiveness and tweak the value of the *prefetch degree* $N$ and the prefetch distance. This prefetcher, like most recent academic designs, fetches into the caches.

> **Takeaway:** The Stream prefetcher concept has seen many academic implementations, with many variations in implementation even though they all prefetch streams of consecutive cache lines.

## 2.4. Industry disclosures

As the industry works behind closed doors, there is often a gap between the academic community publication and what is being shipped in hardware. We will thus also look at what has been disclosed by various actors about their prefetchers.

### 2.4.1. Intel disclosures

#### Early disclosures

In their presentation of the Intel Core microarchitecture, Intel disclosed that those CPUs with a two-level cache contain a pair of L1 and a pair of L2 prefetchers. One of those prefetchers is called the *L2 Stream Prefetcher*. Intel did not divulge the internal details of the prefetcher but indicated it sits alongside the shared L2, alongside a second L2-prefetcher, and that the hardware multiplexes requests of both prefetchers with the requests from the various core private instruction and data L1s. Several bits in Model Specific Register 416 (`0x1a0`) enable or disable some of these prefetchers, as documented by the *Intel 64 and IA-32 Architectures Software Developer's Manual* [47], in Table 2-3, Vol. 4, p. 2-67.

In the Nehalem microarchitecture, Intel introduced a third caching level, with a unified private L2 per core and a shared L3 cache, and documented a Model Specific Register (MSR) controlling four disclosed prefetchers. This register, MSR 420 (`0x1a4`), possesses four usable bits, which have the same description as those in Table 2.1, applicable per core, according to the *Intel 64 and IA-32 Architectures Software Developer's Manual* [47], Table 2-15, Vol. 4, p. 2-141.

#### Prefetchers in the Sandy bridge lineage

In its manuals for the x86 architecture [46, 47], Intel discloses the existence of the same MSR 420 in the Sandy Bridge architecture as in Nehalem, with the same four prefetchers. As of Coffee Lake and Whisky Lake, this register still appeared functional, even though the documentation does not explicitly state it is supported. Its four lower bits each disable one of the disclosed

Table 2.1.: Prefetchers disclosed by Intel for Sandy Bridge CPUs, in [47] Vol. 4 p 2-179.

| Bit | Intel Description |
|---|---|
| 0 | L2 Hardware Prefetcher Disable (R/W)<br>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache. |
| 1 | L2 Adjacent Cache Line Prefetcher Disable (R/W)<br>If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes). |
| 2 | DCU Hardware Prefetcher Disable (R/W)<br>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache. |
| 3 | DCU IP Prefetcher Disable (R/W)<br>If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines. |

prefetchers, the higher bits being reserved. Table 2.1 reproduces Intel documentation of those four bits.

Additionally, Appendix E, Section 2.5.4 in the *Intel 64 and IA-32 Architectures Optimization Reference Manual* [46] gives more details about Sandy Bridge prefetchers, notably calling the L2 Hardware Prefetcher a stream prefetcher. It is stated that these prefetchers may improve performance when data is laid out sequentially but might degrade performance through bandwidth contention when the access patterns are sparse. This is quite consistent with the behavior of stream prefetchers. The worst case documented occurs when a working set is tuned to fit in the L1 and prefetches cause eviction of part of the working set.

Additional details are provided on the L1 cache prefetcher behavior; notably, that prefetch will occur within the same 4 KiB page and does not occur if any fences are progressing through the pipeline, along with a description of both L1 prefetchers, which were not an object of this work.

When it comes to prefetch to the L2 and LLC, it is stated that the so-called L2-prefetchers usually prefetch data into the L2 and the LLC (by inclusivity), but if the L2 cache is heavily loaded prefetch to only the LLC may occur. The *Spatial Prefetcher* attempts to complete 128-byte pairs of cache lines, fetching sibling lines of lines fetched. The *Stream prefetcher* (a.k.a. *Streamer*) reacts to observed L1 misses when it detects sequences of ascending or descending addresses. It may run up to 20 lines ahead in the stream, issuing two prefetches per L2 lookup, and can maintain 32 streams, with a forward and backward stream per page at most.

> **Takeaway:** Intel CPUs of the Sandy Bridge Lineage include four data prefetchers that can be enabled and disabled independently: two L1 prefetchers and two L2 prefetchers. One L2 prefetcher fetches sibling lines of 128-byte pair of lines; the other is a stream prefetcher in ascending and descending directions, with limited implementation details.

#### IP-based Stride prefetcher in Intel Core microarchitecture

In addition to the disclosures relevant to the L2 prefetchers in the Sandy Bridge lineage, Intel made a few disclosures that can enlighten us about the more general design they use. We first discuss their disclosure of the *L1 IP-based prefetcher* in the Intel Core microarchitecture [29]

This prefetcher has a 256-entry table indexed using the lower byte of load addresses. Each entry stores the 12 bits of the latest virtual address requested by a load matching this entry, a 13-bit signed stride value, a 2-bit state machine, and 6 bits identifying the last prefetch. This structure is used to identify striding streams (where consecutive loads by the same instruction are at a constant distance).

The next address required by the stream is then predicted (using the load address to complement the bits not stored in the table) and fetched into the L1 if it is not only present, subject to the limitations mentioned in the manual.

Prefetch requests are issued in a FIFO (with the oldest entries being dropped if the buffer is full) and only issued when the L1 resources are not under use by the demand requests. Prefetchers are also throttled if the bandwidth gets contended.

#### Knights Landing prefetchers

The Knights Landing microarchitecture [110, 111] is the 2015 iteration of the Xeon Phi many-core processor line. They generally use the small execution core design from the Atom line, combined with a mesh interconnect and a specialized memory hierarchy. They pioneered many of the techniques used in the Skylake-SP server processor memory hierarchy. Each core has private instruction and data L1s, while unified L2s are shared between pairs of cores. Intel discloses the presence of an L1 data Prefetcher and an L2 hardware prefetcher, also shared by two different cores. Intel discloses that each L2 prefetcher supports up to 48 streams of consecutive addresses.

> **Takeaway:** Intel has disclosed details of prefetchers other than the Stream prefetch we concern ourselves with; these disclosures may provide hints as to how Intel designs prefetchers in general.

### 2.4.2. IBM Power family

In comparison with the sparse details disclosed by Intel, IBM disclosed many details about its Power CPU microarchitectures, including prefetcher designs. Studying consecutive microarchitectures shows an instructive example of iterative prefetcher evolutions over time, along with the conservation of major principles.

#### Power 4

In their publication describing the microarchitecture of their Power 4 CPU [115], IBM discloses a hardware data prefetcher triggered by streams of L1 misses to consecutive cache 128-byte lines in ascending or descending orders.

This prefetcher fetches lines into the L1 from the L2 and simultaneously prefetches the following addresses of the stream into L2 from L3 and into L3 from memory. After reaching the

stable regimen, a CPU access to the stream's first line triggers the simultaneous transfer of the stream's first line in the L2 cache into the L1 and the first one in L3 into L2. Every four access, a 512-byte line is fetched from memory to L3.

IBM documents that this prefetcher can support 8 such streams simultaneously through an 8-entry table. In order to avoid wasteful stream initiation, the prefetcher slowly ramps up, requiring 4 extra accesses to attain the established regimen, with 1 line in L1, 4 in L2, and 12 lines in L3.

According to the documentation, streams can be initiated in two ways. First, it is possible to explicitly inform the hardware using a specific instruction that a stream should be initiated. Otherwise, the prefetcher guesses the direction of a potential stream on misses. A miss in the lower half of a line suggests an ascending stream, and a miss to the upper half is a descending miss. The entry is confirmed when a miss to the following line in the guessed direction occurs. The prefetcher starts streaming from memory. Otherwise, the entry eventually gets deallocated.

### Power 5

The publication describing the Power 5 CPU [107] indicates that the data prefetcher is retained with minor enhancement and adaptations in this microarchitecture. The prefetcher adapts to the memory hierarchy evolution by fetching the stream's twelfth line into the L2 instead of the fifth. In addition, IBM improved its software-initiated prefetch ability, allowing it to specify the length of a stream initiated in this way. The overall design of the prefetcher is thus conserved from Power4.

### Power 6

This CPU microarchitecture marks a major change compared with the previous out-of-order Power4 and Power5 CPU microarchitectures, as it goes back to an in-order design, using more complex prefetching techniques to avoid the pipeline stalls caused by memory that were previously hidden by the out-of-order execution [57].

One of the features introduced is the Load Look Ahead mode. This mode is entered upon a load miss (or a virtual address translation TLB miss). In that case, the CPU tentatively executes instructions without updating the architectural state and starts any successfully computed loads. When the initial memory request completes, the execution starts back to where it stopped, and instructions are this time executed to completion and their result write-back to the architectural state. These tentative instructions execute with a low priority compared with other threads executed by the CPU.

In addition to implementing an execution-based prefetcher (Section 2.2.2), this CPU also contains an improved version of the stream prefetcher from previous generations. It now possesses 16 separate Prefetch Request Queues and can prefetch 16 distinct streams. It retains the pattern of prefetching into L1 and L2 in a coordinated fashion, with the prefetch in L2 reaching 24 lines ahead. Again the prefetcher will not cross page boundaries; however, it will fully support huge pages (16 MiB in this system).

Additionally, the prefetcher can also prefetch streams of stores. As a result of the cache design, these only need to be fetched into L2, and the signal to detect them is different. Additionally,

store streams are only allocated if there is no matching load stream and get removed if a load stream collides with an existing store stream.

Architecturally, both streams can be triggered by appropriate software prefetch instructions, possibly specifying the stream length. A model-specific register can also control the prefetcher behavior. IBM also discloses some details of the prefetcher ramp-up but does not fully specify it.

### Subsequent evolution

Power7 introduced an ability to prefetch striding patterns. It is, however, disabled by default in the MSR controlling the hardware prefetcher [50].

The Power8 architecture discloses a prefetcher that is apparently similar to the Power6 prefetcher; however, this prefetcher now deals with virtual addresses and is allowed to cross some (but not all) page boundaries [108]. There is also an extension of the mechanism to the newly introduced L4, which receives prefetch information alongside some confidence information [113].

The Power9 disclosure [97] reveals that the prefetcher is now an adaptative design. While the overall architecture of the prefetcher seems unchanged from Power 8, the prefetcher will adapt when the use of prefetched data is low. The Power10 disclosure [114] includes a figure indicating the presence of a prefetcher with 16 stream entries, double the number in Power 9, and suggests the overall prefetcher design is still applicable.

> **Takeaway:** IBM's disclosures provide significant insight into their prefetcher design and evolution over time, which can be enlighting when looking at the evolution of a design in other manufacturers' CPUs.

## Summary

In this chapter, we have seen general concepts about hardware prefetching, including the notion of coverage, accuracy, timeliness, and placement; a general overview of the various class of prefetcher designs that exist; and both academic design and industry disclosure related to stream prefetchers.

# 3. Microarchitecture security

Our computers manage a lot of sensitive data, such as credit card numbers, proprietary information, compromising secrets, cryptographic material used to secure communications, and important, hard-to-replace data such as the result of years of work, family holiday pictures, or documents vital to the operation of a company. Ensuring this data's *confidentiality*, *integrity* and *availability* are thus major imperatives. *Confidentiality* means that the data can only be obtained or accessed by actors allowed to do so; *integrity* means that data is not corrupted and retains its correct value. *Availability* means that the data can be accessed when needed. In addition, computing resources can be expensive and energy-consuming. Therefore, it is also essential to ensure that they are not diverted from their purpose by illegitimate actors.

Generally, abstractions define security guarantees, such as, for instance, process isolation at the Operating System level, which we saw in Section 1.4.3. However, abstractions may be imperfect, and there are many examples of leaky abstractions where such imperfection leads to vulnerabilities, *i.e.*, situations where security guarantees may be violated. Thomas Dullien, formerly from Google Project Zero, summed up as *"(In)Security lives and breathes in the cracks between abstraction layers."*

The field of *microarchitecture security* concerns itself with issues that may arise from the way the microarchitecture implements the architecture. The principal source of leakage is that the architecture makes no guarantee when it comes to execution time and that execution time depends on the actual microarchitectural state.

In our field, the purely software attacker does not have access to the hardware. This is a distinction from hardware security, where the attacker can make physical measurements or interfere physically with the system. Common examples in those cases include respectively electromagnetic emissions or power measurements and injecting faults through, for instance, voltage variations or using a laser.

We consider architecturally correct programs that become vulnerable only when the microarchitecture is considered. However, architecturally incorrect code is still the dominant source of exploits. For example, as of 2019, Microsoft reported that one kind of issue, memory safety, represents over 70% of vulnerabilities discovered, a fraction that has held for over a decade [117].

One last limit to the field we are interested in is that we do not exploit incorrect implementations of the architecture. Violations of the architectural contract by the hardware (e.g., [45, 65]) are outside the scope of our research. One interesting class of vulnerabilities that is thus excluded but is an excellent example of abstraction issues is Rowhammer [53]. This class of attacks arises from the physics of the DRAM technology: repeated accesses to certain memory locations can cause corruption of data adjacent inside the physical DRAM, possibly across privilege domains. This basic primitive can be exploited in various ways, including privilege escalation [100]. As changing architectural values would be against the architectural contract,

our field excludes data integrity and availability issues. *We will thus only explore violations of confidentiality.*

In addition, microarchitecture is both undocumented and has a limited observation surface. The lack of documentation, however, means that mounting attacks usually requires first reverse engineering the architecture. However, this requires observing the microarchitecture itself, which can only be done through the limited observation surface. The primary vector of observation is execution time. Performance counters may give more information, but accessing these is usually restricted to privileged processes. Consequently, attacks and reverse engineering are interdependent processes where older attack primitives are used to further reverse engineer, leading to new attack primitives.

> **Takeaway:** Computer security can be defined as ensuring data *confidentiality, integrity, and availability* and the legitimate usage of computing resources.
> *"(In)Security lives and breathes in the cracks between abstraction layers."* – Thomas Dullien
> In our work, we explore breaches of confidentiality due to the microarchitecture, where correct programs execute on a correct implementation of the architecture but the observable microarchitectural state can leak confidential information. This leakage mainly occurs through timing measurement. Exploiting microarchitectures is a discipline that requires reverse engineering to build attack primitives and using those attack primitives to help reverse engineering.

This chapter will first look at various attack primitives (Section 3.1), possibly eluding the reverse engineering required to design them. We will then cover the reverse engineering aspects, using the primitives exposed earlier (Section 3.2).

## 3.1. Microarchitectural attacks

### 3.1.1. Theory of Microarchitectural attacks

#### The root primitives

The microarchitecture is generally impossible to observe directly by programs. The microarchitecture does not affect the values observed in the architecture, with two exceptions. One of them is that microarchitecture affects *execution time*. Consequently, program execution time and, thus, synchronization between two threads can vary depending on the microarchitecture. The second source of leakage is *performance counters*: registers, usually model-specific, that provide information about the performance of program execution. Those are meant to grant insights into the microarchitecture to understand the issues that may hamper performance.

Regarding the timing approach, most ISAs provide unprivileged instructions that can be used to obtain precise timing information, such as `rdtsc` in the x86 ISA. This is generally sufficient, with judicious use of fences if needed, to measure execution time with enough precision to observe differences due to microarchitectural state. Alternate solutions can be developed for environments where such instructions are unavailable [96], for instance, using a separate thread executing a loop. Fortunately, our research operates exclusively in an environment where `rdtsc` is available.

As for the second approach, performance counters, generally implemented as model-specific registers, can be used to monitor many microarchitectural events. These events may include, for instance, the number of instructions fetched, the number of memory requests handled by a given cache, or the number of misses. These counters are, however, generally unavailable to unprivileged code. As such, they can only be used, in general, in specific attack models, the principal example of which is an attacker with kernel privilege and attacking code running in a secure enclave, such as ARM TrustZone or Intel SGX. This model is not used in this thesis. However, in a reverse engineering context (Section 3.2), the researcher can usually obtain the required privileges to use those counters.

Starting with one of these two primitives, time or counters, attackers generally build higher-level attack primitives to transfer microarchitectural differences to differences observable through the root primitives and from there to the architectural state. Such primitives are, incidentally, another example of abstraction. We will describe in Sections 3.1.2 and 3.1.3 a few examples of such attack primitives.

> **Takeaway:** To transfer microarchitectural state to architectural state, attackers must generally transform the state difference into a timing difference. In specific cases, it is also possible to use performance counters instead. An attack primitive is an abstraction that turns microarchitectural state differences into architectural ones.

#### The two components of an attack

Generally, a microarchitectural attack involves two components. The first component is a microarchitectural structure whose state differs depending on some information the attack wishes to obtain; this is the target of the attack. The second component involved is the primitive used to measure this state and transform the state difference into an architectural value, generally through timing. To support the explanation in this section, we will use the Flush+Reload primitive [135], which targets the cache hierarchy.

Flush+Reload is an attack primitive used to measure the cache state. When an attacker and victim share some read-only portion of the physical memory, the attacker can monitor a given cache line in this area for victim accesses. To do so, the attacker first uses the `clflush` instruction to remove the line from the cache. Any access to the line by the victim will fetch the line into the cache, which creates a microarchitectural state difference. From there, the attacker can then use the time needed to reload the line to distinguish the two microarchitectural states.

In this attack, the targeted structure is the cache, and the attack primitive is the combination of `clflush` and a load instruction used to re-load the line, hence the name of the attack primitive. We will see in Section 3.1.2 other attacks that target the cache.

It is worth noting the method may include constraints on the placement of the attacker and victim on a multicore system. For instance, some attacks require both to execute on the same core, one after the other (*same-core*). Others require that the attacker and victim run simultaneously on two SMT threads of the same physical core (*SMT sibling threads*) or run on different physical cores. Finally, others may have more relaxed constraints (*multicore* techniques).

> **Takeaway:** A microarchitectural attack consists of two components: a microarchitectural target structure and a method to turn differences in the microarchitectural structure state into architecturally observable differences, chiefly execution time differences.

### Side and Covert channels

We have seen that an attacker can monitor memory accesses by a victim to a given read-only cache line over time. How can this be used to violate confidentiality, given that the shared read-only data shared is not confidential? Two configurations exist.

In the first case, two processes collude to attempt to exfiltrate information, for instance, a process that has access to confidential data but no access to the network and another process that may transmit data over the network but does not have access to the confidential data. If those two processes can share a portion of read-only memory, the attack primitive can be used to transmit information by encoding it in a sequence of memory loads to a shared cache line. This is an example of a covert channel, which corresponds to cooperation by attacker-controlled processes in two distinct security domains to transmit information that should not be.

Most covert channels are exposed to sources of errors. Consequently, estimating the *true capacity*, which corresponds to the bandwidth once the required error correction is added, is essential. It can be estimated using the raw bit rate of the channel $C$ and the error rate $p$ of the transmission, following the following formula: $T = C \times (1 + p \log_2 p + (1 - p) \log_2(1 - p))$ [81].

This first configuration can be distinguished from a side-channel attack. In this second configuration, the attacker controls only one process and tries to extract confidential data from an unaltered victim process. The Flush+Reload technique can extract information from a process if that process's sequence of memory accesses to shared pages depends on confidential data.

In either case, memory sharing can be realistic, given that OSes load libraries into physical memory once and map this one copy into every process requiring the library as shared read-only pages. The code is usually immutable, and the OS copies mutable data in the physical address space when a process attempts to modify it, a technique called Copy on Write (CoW)[1].

Given this pattern of sharing the immutable code and read-only data, code patterns such as branching depending on a secret value or indexing into a read-only structure using confidential data can lead to such leaks. Sensitive libraries, among which cryptographic libraries such as OpenSSL, usually try to write code that does not induce microarchitectural state differences depending on secret values, a practice generally referred to as *constant time*.

[1] This sharing occurs at the physical page level, even when the shared pages are mapped at different virtual addresses.

> **Takeaway:** One generally distinguishes two situations for microarchitectural attacks: *side-channel attacks* when an attacker eavesdrop on a non-cooperating victim process, and a *covert channel* when two processes in different security domain collude to transmit information across a boundary where they should not be able to. Reliable covert channels may be established over unreliable channels, and their true capacity $T$ can be estimated as $T = C \times (1 + p \log_2 p + (1-p) \log_2 (1-p))$, with $C$ the raw bit rate and $p$ the error rate.

### Transient and Persistent state attacks

Another distinction regarding microarchitectural attacks that should be made is whether the difference caused is part of some *persistent* state or only occurs during a short window of time. For instance, modifying whether a line is in the cache is persistent, at least until enough memory accesses have occurred to alter the state. Flush+Reload is thus an attack that targets persistent state.

In contrast, modern CPUs include many shared resources that can be contended for different processes. As an example of a technique called *port contention* [2], an SMT thread running on a CPU with a single floating point division unit can detect whether a sibling thread is executing floating point divisions by observing the throughput achieved on a loop repeatedly executing floating point divisions. This *contention* attack is an example of *transient state* attacks.

Port contention attacks are an interesting example as they can be exploited in many contexts, provided the attacker and victim can be located on the same physical core. This is because this attack requires no special instructions aside from a timer primitive and no shared memory. It has thus been exploited in set-ups like web browsers [95].

Transient state attacks require precise simultaneous execution of the attacker and victim process, while persistent state attacks may work even if they do not execute simultaneously.

Persistent state attacks, but not transient state ones, can be decomposed in three phases:

1. First, the attacker sets up the microarchitectural structure to a known state.
2. The victim causes changes in the microarchitectural structure
3. The attacker converts this difference into an architectural difference, which is usually called observing the difference.

> **Takeaway:** Microarchitectural attacks can be distinguished into two categories, *transient* state and *persistent* state attacks. Transient state attacks require simultaneous execution of both victim and attacker and result in observable behavior due to the behavior of some component when requests from two threads come in simultaneously compared with a single one. Persistent state attacks, on the other hand, follow a three-step process, with component set-up, victim change to the component state, and observation of the changes.

### Transient execution attacks

*Transient execution* attacks are a class of attacks that has significantly developed since their discovery in 2018. Those attacks exploit speculative execution or faults in modern CPUs, which

may result in transient violations of security guarantees. More precisely, transient execution attacks arise from the tentative execution of instructions that will eventually be discarded. This can occur either as the result of a prediction, e.g., a branch prediction, where instructions get discarded if the prediction was incorrect; or as the result of a fault when the CPU detects the fault late in the pipeline and allows later instructions to proceed and possibly exploit a result of the faulting instruction. These sources exist even in in-order pipelined CPUs, but out-of-order CPUs expose a much larger window of transient execution. In either case, the code executed transiently manages to obtain information otherwise inaccessible and encodes it using a microarchitectural channel. Persistent execution then reads from the covert channel to recover the leaked data and transfer it to the architectural state. The most common way this is done is using a Flush+Reload channel, encoding the illegally obtained value in the cache state by bringing a specific line into the cache depending on the leaked value. Many of these attacks have made their way into the general press, such as Meltdown [62], Spectre [54], Fallout [18] and others.

This thesis does not involve transient execution attacks; hence we will not cover the details of the various transient attacks. However, covert channels, which we study, are used as part of such attacks.

Transient execution attacks can be divided into 6 steps [35]:

1. First, the microarchitectural state is prepared for the attack. This step is similar to the set-up in conventional microarchitectural attacks.
2. Then transient execution is triggered, either in the victim (confused deputy) or the attacker code.
3. The transiently executed code accesses the secret data, violating the architectural security guarantees.
4. The transiently executed code transmits the secret through a microarchitectural side channel that survives past the end of transient execution.
5. The hardware detects and recovers from the mis-speculation and fixes the architectural state, erasing any incorrect architectural state resulting from the incorrect transient execution.
6. The attacker recovers the secret encoded in the microarchitectural state.

It is possible to use a transient channel instead of a persistent one for steps 4 and 6, in which case 6 must run concurrently with 4.

We will provide one concrete example of a recent attack of this class, Retbleed [129], and refer the reader to recent surveys [17, 132] or Gruss's habilitation thesis [35] for more details.

Retbleed is a transient execution attack that misdirects the execution on Intel and AMD when the program encounters a return instruction (`ret`). It impacts the CPUs from Intel and those from AMD differently. However, in either case, the execution is misdirected to execute a *confused deputy*, as in Spectre-BTI [54]. This misdirection primitive allows redirection of execution to an arbitrary location in the victim program, which is unwittingly used to leak information. A commonly used gadget accesses an array using an attacker-controlled offset and then uses this value to index into an array visible to the attacker. Nevertheless, many other

possible gadgets can be used if they can be found in the victim program. The security violation is the out-of-bound access in the first part of the gadget, and the gadget's second part encodes the value using a cache side channel.

On Intel CPUs, Wikner et al. found out that the Branch Target Buffer (BTB) would be used to predict future execution upon encountering a `ret` instruction if the return stack is empty. This configuration can happen when the Return Address Stack has been previously overflowed, causing it to drop its oldest entries. The paper's most significant contribution is identifying exploitable control flow graphs in the Linux kernel, which can allow this situation to occur.

On AMD CPUs, they showed that if a matching BTB entry exists for a `ret` instruction, or, even, *any* instruction, it overrides the next instruction prediction, including that from the RAS (*return address stack, c.f.* Section 1.2.4), and causes execution to be diverted. As a result, it is possible to misdirect execution very easily on AMD CPUs. The only constraint is that the history of recent branches matches that used to train the entry.

In both cases, this defeats defenses integrated into the kernel to mitigate this Spectre variant, removing branch instructions predicted by the BTB and replacing those with return instructions, a technique named retpoline.

> **Takeaway:** *Transient execution* attacks are more complex and powerful microarchitectural attacks that abuse the execution of instructions that later get discarded, called *transient execution*, and combine it with the usual microarchitectural channels. Transient execution attacks cause the CPU to execute transiently incorrect code, either through branch mis-prediction or faults. Then, in the transient execution domain, they violate the security guarantees to obtain a secret and encode it on a microarchitectural channel. Finally, the attacker extracts the secret from the channel after the hardware recovers from the incorrect execution.

### 3.1.2. Cache attacks

Generally, *cache attacks* designate attacks that exploit the status of lines in the cache to obtain information, such as whether a line is present or absent, or in some cases, more involved cache coherence or metadata state. Generally, this allows an attacker to obtain information about addresses accessed by a victim program.

There are roughly two models for these attacks, one where shared read-only memory is targeted and one where no memory sharing is possible. Flush+Reload and Flush+Flush belong to the former model, while the latter model generally relies on addresses mapping to the same set (often called congruent addresses); this is the case of, for instance, Prime+Probe.

#### Flush+Reload

This is the archetypal cache attack, which we have used as an example throughout this section. To summarize, this attack uses the `clflush` instruction to set up the cache line used so that it is absent from the cache. The victim then executes, possibly bringing the line into the cache, and then the attacker measures the presence of the line by timing a load to this specific line [135]. Flush+Reload is relatively fast — the set-up is a single instruction, as is the measurement — and accurate — the distinction between a hit and a miss is pretty apparent when looking at

the timing of a load. It is worth noting that thanks to the property of `clflush`, this attack only requires the attacker and victim to share an inclusive cache, which is the case of many Intel CPUs, including the entire Sandy Bridge Lineage.

Gruss et al. [37] developed an automated framework for such attacks. Using this framework, they developed a chosen plaintext attack on AES implementations using T-tables, which we will use as an example later on (Section 5.5.2). Many attacks on the AES primitive have been developed, and it is neither the first [11, 38, 55] nor the last. T-tables are a way AES can be implemented in software. It uses large look-up tables to implement the AES rounds. This implementation is particularly susceptible to cache attacks as the table lies across 4096 B, representing 64 cache lines. Consequently, many publications [1, 4, 5, 14, 37, 38, 127] have used this vulnerable implementation as a benchmark for their cache attack, and so will we.

> **Takeaway:** Flush+Reload is the archetypal cache attack, which requires shared memory and a shared, inclusive cache, providing a fast and accurate cache channel at the cache line granularity.

### Flush+Flush

Gruss et al. [36] showed that, on Intel CPUs, the `clflush` instruction takes a different time to execute depending on whether its target cache line is cached or not. This timing difference is around a dozen cycles, significantly smaller than the difference between a load hit and a load miss. However, on average, `clflush` executes faster than a load, with `clflush` of an Invalid line (`clflush` miss) being faster than `clflush` of a line present in the cache (`clflush` hit). From this observation, the Flush+Flush attack primitive is built by replacing the reload step in Flush+Reload with a timed `clflush`, which simultaneously measures and resets the state of the line. Flush+Flush is thus a faster attack than Flush+Reload.

Furthermore, the attacker thread encounters no cache misses, unlike Flush+Reload attacks. As some approaches for cache attack detection use performance counters to detect abnormal rates of misses, this makes Flush+Flush faster and stealthier than Flush+Reload. However, this comes at the cost of some accuracy, which we will investigate in Chapter 5. Another source of timing variation that impacts Flush+Flush significantly is the scaling of the CPU frequency, for which Saxena and Panda [98] have proposed a solution if DVFS is not disabled.

Thanks to `clflush` operating on the entire cache coherence domain, which in modern Intel CPUs, covers all sockets in the system, Flush+Flush, in theory, has more flexible requirements than Flush+Reload and may work on systems with non-inclusive caches.

> **Takeaway:** Flush+Flush is a fast and stealthier version of Flush+Reload, albeit a more noisy one, that leverages the difference in execution time of `clflush`, depending on the caching status of its operand. This attack is at the core of Chapter 5.

### Prime+Probe

We have seen two attacks that rely on shared memory; however, this assumption is not always valid. Consequently, we will also show an attack that can work without that assumption. This earlier attack from 2006, described by Osvik et al. [84], requires neither shared memory nor an

instruction to evict lines. Instead of easily evicting the specific memory address used by the victim, it uses the limited associativity of the cache to cause a conflict miss on that address by accessing a sufficient number of different congruent addresses, *i.e.*, addresses in the same cache set. A sequence of accesses that can be used to evict a line from a set is named an *eviction set*. This attack notably applied to AES [84] following a series of other earlier attacks on this cipher [11, 55]. One situation where this attack is applicable is cross-VM attacks [63], as those usually do not share memory.

In practice, the Prime step is the set-up of the attack, filling a cache set with the attacker's eviction set. Then, when the victim requests a congruent address, it causes the eviction of one of the attacker's lines. The observation step, named the Probe stage, then consists in timing accesses to each line in the set, detecting if any of these got evicted.

In platforms where an instruction like `clflush` does not exist, such as on ARM versions before v8, eviction sets were the only technique usable, with the added difficulty that those CPUs may have pseudo-random eviction policies and other features that hamper the attack [34]. However, cache attacks are still possible on such platforms [59].

> **Takeaway:** Prime+Probe is a generic attack technique usable with fewer assumptions than attacks based on shared memory, albeit slower and less precise. It relies on eviction sets to detect accesses to congruent cache lines.

#### Other cache-based primitives

Many other primitives have been developed using caches to handle subtle variations in the attacker assumptions. We will give a few examples of such primitives, with an example dedicated to non-inclusive caches, an improvement on Prime+Probe that can obtain a more precise temporal resolution provided the cache eviction policies are well understood, and a clever utilization of software prefetch instructions. However, those attacks are conceptually similar to Flush+Reload and Prime+Probe.

Yan et al. [134] showed that in systems where the last-level cache is non-inclusive, it is possible to target the cache coherence directory, which must contain entries for each line located in the cache hierarchy. They first reverse engineered the cache directory structure and determined that they could build eviction sets for the directory.

A more refined understanding of the eviction policies used by inclusive last-level caches and cache coherence directories led Purnal et al. [92] to propose the Prime+Scope attack. This attack works by setting up the last level cache or cache directory so that a specific line is the eviction candidate but remains available in a lower level private cache. Any access to a congruent address will cause eviction of this cache line. The measurement step measures the presence of the cache line in the lower level cache, which does not influence the state of the higher level shared structure. If the line was evicted from the shared structure (inclusive LLC or cache directory), it must also be removed from any private caches, causing an observable miss for the attacker. Earlier, Briongos et al. [15] built the Reload+Refresh attack that avoids causing misses to the victim, whereas Flush+Flush avoids causing misses for the attacker, leveraging the eviction policies of certain Intel caches.

One last primitive that has been found is a surprising behavior of the `prefetchw` instruction,

observed by Guo et al. [39]. On Intel CPUs, the `prefetchw` instruction ignores the actual permission bits and puts its operand in a state suitable for modification, the M state, according to the authors, even when the cache line is read-only. Even though store instruction to this line would fault, preventing any architectural violation, this property of `prefetchw` is exploitable to mount cache attacks. First, a cross-core covert channel can be built where transmission is done by prefetching or not, and reception can use a load to determine if the coherence state was changed to Modified by another core. Moreover, a side channel can also be built by using two different cores, one of them prefetches the address, and the other reloads it, detecting whether a third core requested it. Differences in the loads' execution time will reveal whether a third core, the victim, requested the line and changed its state back to Shared. Finally, it was found that, on these CPUs, the `prefetchw` took a different time to execute depending on the coherence state, which allows building Prefetch+Prefetch similarly to Flush+Flush.

More generally, cache attacks on cloud computing and virtualized environments [93, 130, 133], were shown to be practical threats [63, 66]. Maurice et al. [68] also studied protocols that could obtain a reliable channel on top of various covert channel primitives. This showed that even noisy covert channels could be used to leak information reliably.

> **Takeaway:** There are many cache attacks with various assumptions, which generally can be understood as working similarly to Flush+Reload or Prime+Probe, possibly with better speed or accuracy. Additionally, such attacks are practical on virtualized set-ups and even across the network.

### 3.1.3. Other attacks targets

Caches have been the most prominent target for microarchitectural attacks; however, these are not the only possible targets. We will illustrate here a few other existing attacks.

#### Branch prediction state

The first target that we will present is the branch prediction logic. In practice, current CPUs do not tag branches with unique information, and it is possible to cause different branch instructions to match the same entries in one of the tables used. The BranchScope attack [31] is an example of such an attack: it targets the branch direction predictor, used to predict if a conditional branch will be taken or not.

Thanks to some reverse engineering, this work found that they can set up a branch entry in a state where observable differences appear depending on whether a congruent victim branch was taken. To be precise, the attacker branch's execution time discloses the direction the victim branch takes. Other attacks have been mounted [12] on the various branch predictors, and Spectre [54] leverages the ability to mistrain branches to cause speculative execution.

> **Takeaway:** Besides its use in the second step of a transient execution attack, misdirecting speculative execution, *branch predictors* are components susceptible to microarchitectural attacks leaking the direction taken by conditional branches.

### Contention attacks

As seen in Section 3.1.1, observing differences in execution time due to SMT threads sharing resources is possible. For instance, this has been exploited to build port contention attacks [2, 95], exploiting constraints on the set of functional units on which each instruction can be dispatched and the ability to cause contention of these execution resources.

A different contention attack has also been previously proposed, *CacheBleed* [137], which exploits a limitation of the cache architecture in the original Sandy Bridge L1D cache. These limitations were later fixed in the following iteration of the lineage. The Sandy Bridge L1D was divided into two banks, and it was possible to design access patterns that would cause contention when both the victim and attacker access the same bank.

One more recent work, the SQUIP attack [33], showed that it was possible to cause contention in the queue structure used by the CPU to schedule instructions. In their case, they specifically targeted a system that has separate queues for different types of instructions, and filling up one queue causes the stage that dispatches the various instructions in the queues to stall, preventing the dispatch of instructions that would go to other queues. This can then detect if the sibling thread executed an instruction belonging to a specific queue.

These three examples of contention attacks work with attackers co-located on a physical core. However, this is by no means a requirement for contention attacks. We will see in Section 3.2.1 a different contention attack that can be used with attackers on separate cores that is note-worthy for its reverse engineering.

> **Takeaway:** Causing contention on various execution resources is a viable strategy for attacks with many different targets.

### Power consumption and DVFS

One very recent work [126] showed that the difference in power consumption caused by differences in the data values used in computation could cause observable differences in the Dynamic Voltage and Frequency Scaling selected by the system. Under some circumstances, those differences are discernible from the network, a vulnerability named HertzBleed by its authors.

This follows the Platypus attack [61], which found that Intel CPUs exposed an unprivileged interface to power consumption measurements in the system and used it to break various security protections and infer data flowing through the system, extracting AES keys manipulated using the hardware AES-NI instructions from secure enclave and kernel alike, and again breaking kernel ASLR.

In either attack, the Hamming weight of the values manipulated in the system directly relates to the power consumption. This fact is not unexpected given the CMOS technology, but showing that this creates a signal observable from software is a significant breakthrough.

> **Takeaway:** Power consumption is also a target that can lead to exploitable signals for software-based attacks.

### The "Pandora's box" theoretical study

In addition to the attacks developed above, Vicarte et al. [121] noticed that a large number of microarchitectural techniques had been proposed in the academic literature but had not been evaluated from a security point of view and had not been found yet "in the wild," that is, in shipping products. Consequently, they studied seven different microarchitectural techniques proposed in the literature, evaluating their security impact.

The most striking example in these findings is that many designs of indirect prefetchers or data-dependent prefetchers could leak as much memory as the Meltdown attack [62], a universal read transient execution attack. To be precise, they identify that any prefetcher dedicated to prefetching `A[B[C[stride*i]]]`, or an equivalent structure with pointer indirection, could be turned into a read primitive when the attacker can append after a valid array C the address of a victim address.

They showed that techniques simplifying computation and compressing the data in the pipeline could lead to transient state (or stateless) attacks. Meanwhile, techniques such as silent stores, reusing computations, and value prediction could be used for persistent state (or stateful) attacks. Lastly, register file compression and data-dependent prefetchers can also lead to significant attacks using the architectural state.

> **Takeaway:** Many microarchitectural optimizations can be turned on their head to become microarchitectural leaks. While academic publications can be reviewed and analyzed, the obscurity of modern CPUs may hide similar issues.

### Summary

In this section, we have covered the basic principles of microarchitectural attacks, including the classification of attacks, and the various steps of the attack process, before seeing examples of both cache-based persistent state attacks and a variety of *persistent* and *transient* state attacks.

## 3.2. Microarchitectural reverse engineering

Most attacks discussed above rely on understanding the microarchitecture to find ways of transferring microarchitectural state differences to architecturally observable differences. However, the documentation of the microarchitecture is sparse and does not contain the details required to mount attacks. Consequently, security researchers need to figure out those details by themselves, designing and running appropriate experiments.

In this section, we will look at a few examples of reverse engineering, including results on top of which we built our research, concurrent work that corroborates our research, and other enlightening examples of reverse engineering. While most of these works also led to attacks, demonstrating the value of the reverse engineering work, these attacks are less in our focus here.

There are a few general principles that can apply to reverse engineering hardware components. A first step is generally to collect a significant amount of data, using either timing measurements or performance counters. Once this is done, the data has to be analyzed to identify a model,

which can then be verified to predict the system's behavior. In addition to brute-force approaches, functions involving addresses, such as memory addresses, can often be uncovered by reducing them to linear algebra equations and solving those. Most other structures have generally required ad-hoc strategies.

### 3.2.1. Cache reverse engineering

Because caches are one of the most used structures for microarchitectural attacks, they have seen a lot of studies to obtain a detailed understanding of their implementations.

#### Intel cache slices

As we have seen in Section 1.5.2, the L3 cache in the Sandy Bridge lineage of CPUs is divided into slices, with physical addresses distributed over *slices* using an undocumented hash function. Several approaches have been used to reverse engineer those slices. An initial assumption that was taken and later verified is that the offset bits of the cache line are not involved in the hash function, as those would otherwise cause different parts of the cache line to map to different slices, which does not make sense. It was found that this hash function is linear with respect to the bitwise exclusive OR ($\oplus$) when the number of cores in the system is a power of two. Two classes of approaches were used, cache set conflicts and performance counters, with different trade-offs.

Using the conflict approach, we identify addresses within a set that conflict and, therefore, map to the same slice. This approach only finds conflicting addresses with the same set bits, and the influence of those bits is unknown. Consequently, while it cannot identify exact slices, it provides sufficient information to construct sets of congruent addresses on those caches and enable Prime+Probe types attacks. This approach was pioneered by Hund et al. [43] and further refined by both Irazoqui et al. [48] and Yarom et al. [136]. However, this approach will be able to deal with microarchitectures with several slices per core, such as Skylake and its derivatives.

On the other hand, the performance counter approach [67] obtains information from which core serves requests for a given address and can thus get absolute information on slices, instead of the relative *conflicts-with* information obtained by the other methods. In addition, it can investigate the set bits, demonstrating that the set bits matter in most cases. However, it depends on having access to the appropriate performance counters and thus requires privileged access.

For systems where the number of cores is not a power of two, a non-linear component must be introduced to split the address space into parts of close but not identical sizes [136], the reverse engineering is generally a bit more ad-hoc than for powers of two.

We expose more details about those slicing functions in Appendix A, including the detailed expression of the hash functions used on most systems from Sandy bridge to Coffee Lake, where the number of cores is a power of two, along with a pair of new functions uncovered as part of the research in Chapter 5.

The line of server CPUs started with Skylake-SP uses a different core interconnect and non-inclusive caches but still uses the same hashing principle. McCalpin has released reports [69, 70] exposing methods and results applied to several of these CPUs.

> **Takeaway:** The hash functions used to split the address space over the slices of the last-level cache in Intel caches can be deduced from cache set conflicts or performance counters. Each approach has its own trade-offs.

### Reverse Engineering Eviction policies

We have just seen how to identify which addresses are congruent. However, to build finer-grained attacks and properly simulate cache operations, it is also essential to understand the exact eviction policies used, which are usually not rigorous Least Recently Used, as implementing this policy is expensive.

Thus, a line of work has been dedicated to reverse engineering eviction policies used by caches. One major contributor to the field is Vila, who, throughout his thesis [122], built a complete framework to do so.

The first component of this work was an approach to systematically find minimal eviction sets, an important primitive used to control cache state [124]. This is then used as a primitive by the CacheQuery framework [123]. This latter framework comprises three layers, with different levels of abstraction that allow learning the eviction policies used by a given cache set by checking if given sequences of accesses resulted in misses or hits.

> **Takeaway:** Reverse engineering efficiently and systematically usually involves designing a conceptual and practical framework, which may contain several layers of abstractions.

### The cache interconnects

In work concurrent with ours [26] (see Chapter 5), Paccagnella et al. [85, 86] investigated the low-level details of the core interconnect in the Sandy Bridge lineage of CPUs. Building on top of the method from Maurice et al. [67] to reverse the slice hash function, they first show the dependence of load time for LLC hits depending on the slice where the cache line belongs.

From there, they run an experiment with two threads to detect configurations in which contention occurs, with one thread creating heavy memory traffic to a given slice and a second one measuring the latency of loads to each slice. In addition to observing contentions when both threads target the same slice, they identify cases where those two threads contend while targeting different slices, a contention only the ring can cause. The exact circumstances under which contention occurs allow them to infer the system's topology shown in Fig. 3.1 and the following observations on the ring structure and communication protocol. Traffic flowing in opposite directions does not interfere, and loads travel through the shortest path. Traffic cannot interfere if the segments do not overlap. Traffic within the ring is prioritized. Consequently, contention occurs when traffic on the ring prevents the injection of traffic from a slice or a core. The System Agent (connected to DRAM) and GPU are also nodes on the ring that can contend when injecting traffic. Each ring has two lanes, and nodes, both cores and slices, are partitioned into two different sets. Each lane is dedicated to traffic to a specific set of nodes. A core and its associated slice are part of different sets. Cache misses still traverse the ring to the LLC before being sent to the system agent, who then responds directly to the core. Additionally, the LLC

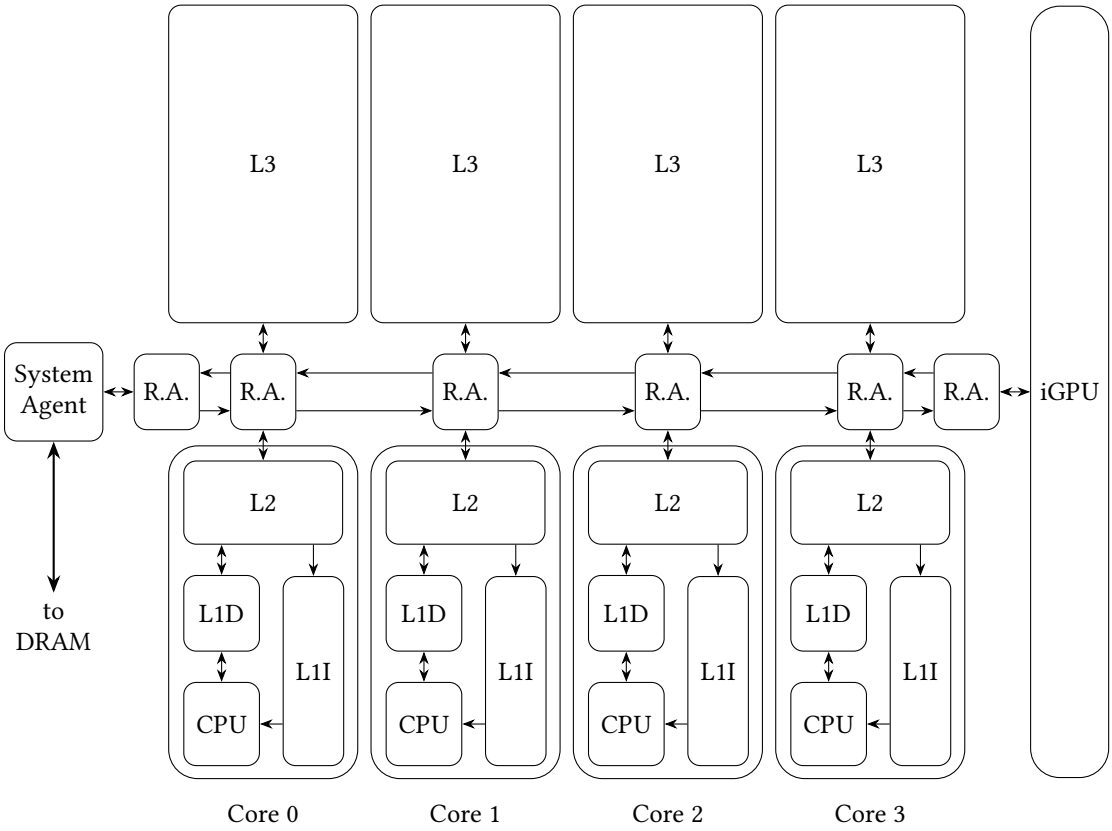



Figure 3.1.: Topology of the interconnect in the Sandy Bridge Lineage of CPUs

acknowledges the core request and gets a copy of the loaded data.

Overall, this massive amount of findings was obtained by observing contention for many cases of hit and misses. This allowed Paccagnella et al. to build a high-bandwidth covert channel and a novel contention side-channel, applicable on Coffee Lake and Skylake CPUs and likely applicable to all CPUs in Sandy Bridge Lineage featuring a single ring. Further work from the authors in the extended version showed the technique's applicability to the server CPUs of the lineage that use two different rings.

Further work by the same group [25] investigated Intel's more recent server CPUs, which use a *mesh* interconnect. Using the same techniques, they inferred the protocol used by this more complex interconnect, which uses different policies for its vertical and horizontal rings. Again, this allowed them to build covert channels and side-channel attacks.

> **Takeaway:** A study concurrent with ours reverse engineered low-level details of the Sandy Bridge lineage's L3 cache and inferred its interconnect topology. They also uncovered general principles of the communication protocol used.

#### AMD µTag predictor reverse engineering

Since the Bulldozer Architecture (2011), AMD CPUs use a way-predictor in the L1D cache to predict which cache way will be accessed. This then allows energy saving by only activating the corresponding way. This predictor first hashes the virtual address to obtain a *µTag*. This tag is then used to look up the predicted way for the access from a table. The corresponding way is activated, and if the tag matches, an early hit occurs, with energy savings. Otherwise, the other ways are activated and checked. Addresses that map to the same µTag induce conflicts that thus delay memory access to one another.

Lipp et al. [60] uncovered the hash function used and the characteristics of the cache-way predictor in those AMD CPUs from 2011 to 2019. Working from the assumption that the hash was a linear function, they used the delay caused by conflicts to identify which bits were used in the hash function and reconstructed it, verifying the linear assumption. They then ran experiments to determine how this structure was shared in SMT contexts and were able to construct new cache channels using these structures in a sibling hyper-thread context.

> **Takeaway:** Overall, caches have been a prominent focus of reverse engineering efforts, thanks to the significant timing differences between cache hits and misses and the solid primitives used to support this research.

### 3.2.2. Prefetcher Reverse Engineering

As we have seen in Section 2.4, manufacturers document sparsely any prefetchers included in their CPUs, assuming the existence of those prefetchers is even disclosed. Consequently, prefetchers have been the target of significant reverse engineering, especially given that they have the potential to be exploited in microarchitectural attacks [121] as we saw in Section 3.1.3.

#### Intel L1D IP-Stride prefetcher

In addition to Intel's more substantial than usual disclosure on the topic, several different teams [19, 24, 105] have studied this prefetcher, revealing various security implications. Shin et al. [105] used the Flush+Reload primitive and showed the prefetcher exhibited some amount of non-determinism but caused leakage in otherwise constant time code. Depending on the order of the memory accesses, the prefetch may detect a striding stream and fetch cache lines that otherwise would not. For instance, Cronin et al. [24] showed that the prefetcher used a table of entries that was conserved over context switches and could be used to build a covert channel. They identified that three accesses were sufficient to create a striding stream. Chen et al. established that this prefetcher state is shared by sibling SMT threads, which enables sibling SMT side and covert channels.

> **Takeaway:** There has been a significant amount of work on the L1D IP-based stride prefetcher in Intel CPUs

#### Intel Stream prefetcher

A first study of the L2 Stream prefetcher on Kaby Lake CPUs was undertaken by Rohan et al. [94]. They confirmed through reverse engineering many of the prefetcher's properties described in Intel's documentation [46]. Precisely, the prefetcher issues requests to consecutive cache lines in the positive (increasing addresses) or negative (decreasing addresses) direction. It is possible to start a stream and then trigger prefetches with an access further away in the same page. Their estimate that the prefetcher can track streams in 16 pages was consistent with the documentation, which indicates that a positive and negative stream can be tracked per page for a total of 32 streams The prefetcher state is shared between hyper-threads, and they were thus able to build a covert channel over it. Lastly, they showed it does not operate across 4 KiB page limits, even with 2 MiB huge pages.

> **Takeaway:** There has been a more limited amount of work on the L2 prefetcher. While we have indications on the size of the table used and that it does not prefetch beyond 4 KiB page limits, we do not have more precise details.

#### Prefetcher reverse engineering on Apple M1

In work concurrent with ours, Vicarte et al. studied prefetchers in the Apple M1 CPU [120], a custom ARM design for general purpose computers scaled up from the Apple Ax CPU used in iPhones and iPads. Their reverse engineering showed that this CPU features a data-dependent prefetcher fetching pattern of the form `*A[stride*i]`. As this prefetcher only prefetches a single level, it is not vulnerable to the theoretical attack exposed in 2021 by Vicarte et al. [121]; however, it is exploitable to break Kernel Address Space Layout Randomization and can leak some amount of data within constraints. This reverse engineering effort was built on top of a cache eviction primitive and load timing.

> **Takeaway:** A data-dependent prefetcher has been found in Apple CPUs, with some significant but not major security impact.

### 3.2.3. Other components

#### Branch predictors

Uzelac et al. [118] proposed a methodology to reverse engineer branch prediction units. On the Intel Pentium M, they ran micro-benchmarks that revealed the branch predictor structure based on their execution times. More recently, Bhattacharya et al. [12] reconstructed the branch direction predictions of Intel CPUs of microarchitecture from Nehalem to Broadwell and showed to be consistent with 2-bit (Nehalem) or 3-bit saturating counters. This was done using the insight from performance counters. The resulting knowledge was then used to mount a side-channel attack on blinded scalar multiplication. In addition, Spectre v1 [54] relies on knowledge of the branch predictor, as does the BranchScope [31] attack, mentioned earlier (Section 3.1.3).

### Memory system

Pessl et al. [91] devised two different methods to reverse engineer the DRAM addressing schemes and used these to build cross socket channels requiring no shared memory. Their insights also improved the ability to run Rowhammer attacks on DDR4. The first of those two methods used physical monitoring of the communications between the CPU and DRAM. In contrast, the second method runs entirely in software and is another example of reverse engineering efforts. It uses timing differences in DRAM access time due to what is called row conflicts to recover data and then solves a system of linear equations. The Rowhammer physical effect was also used on systems with ECC to cause information leakage across DRAM rows, an attack known as RAMBleed [56].

Van Schaik et al. [99] uncovered page-table caches, which are used to speed up page-table walks on CPUs from Intel, AMD, and ARM. They developed the RevAnC framework to reverse engineer those caches, which are consulted after a TLB miss but before accessing the data caches. More precisely, these caches store partial translations, which can significantly speed up TLB misses handling.

> **Takeaway:** The same general principles of reverse engineering apply to many other components in computer implementation.

### Summary

In this section, we have covered the general principles of microarchitecture reverse engineering and looked at efforts targeted at various components, with a greater focus on caches and prefetchers. We have seen that the gained understanding could often result in new microarchitectural attacks.

## Conclusion

In this chapter, we have looked at how reverse engineering and microarchitectural attacks are intertwined. From the root primitives, execution times and performance counters, more complex primitives can be built to obtain insight into the microarchitectural state. An attacker can then use this insight to exploit microarchitectural leaks while these insights are used in reverse engineering to collect information and better understand the system.

Among the various works we covered in this chapter, the Flush+Reload and Flush+Flush techniques are essential to our work, exposed in Part II, as are the reverse engineered knowledge about Intel L3 slices and L2 Stream prefetcher and the concepts in Chapter 1. Furthermore, the prefetcher design knowledge in Chapter 2 is essential to understand the work exposed specifically in Chapter 6.

This chapter thus closes the background, and with the aforementioned important takeaways in mind, we can now move on to the research results of these three years of work.

# Part II.

# Research results

# 4. Motivation and General considerations

In this chapter, we discuss some of the overall motivations and goals of the research work, which led to the two conference publications discussed in Chapters 5 and 6. Section 4.1 explains why we decided to investigate prefetchers, Section 4.2 explains why we chose to use Flush+Flush, and Section 4.3 describes our initial difficulties. This last section sets the stage for Chapter 5 and elucidates why this first paper does not discuss prefetchers. Section 4.4 will also introduce the original goals of the framework we started to build at that time and developed and extended over the three years of this thesis. Lastly, we present the hardware and software configuration used in this research in Section 4.5.

## 4.1. Why study prefetchers?

The original goal with which I set out for this Ph.D. was to contribute to the area of microarchitecture security. It came to my attention during a guest lecture by an Apple Engineer that prefetchers were undocumented and could potentially violate security assumptions in unexpected ways.

### 4.1.1. Lack of documentation

Prefetchers lie entirely within the microarchitecture and are only observable through their impact on caches, another microarchitectural component. Consequently, manufacturers only need to disclose very little about them. In addition, manufacturers typically use these components to gain an edge over their competitors; hence there is a strong incentive not to give away any details. We saw in Chapter 2 how manufacturers remain vague about aspects of their prefetchers.

However, microarchitectural research, as we have seen in Chapter 3, needs to peek under the hood, as security issues can also be found at the limits of abstractions when such imperfect abstractions leak aspects of the underlying implementation. There is thus a solid incentive for security researchers to understand prefetchers.

### 4.1.2. Potential security impact

Prefetching is a pretty speculative activity, and at the time I started this Ph.D., the first transient attack had just been published. At the time, it was an open question whether prefetchers did anything that could be unsafe or violate security guarantees in a way that would cause information leaks.

As later shown by Vicarte et al. [121], some theoretical prefetcher designs can be turned into an arbitrary read primitive. This is the case, for instance, of indirect memory prefetchers with

at least two levels of indirection. As mentioned in Section 3.2.2, a data-dependent prefetcher was later found in CPUs designed by Apple, with a security impact. Luckily that prefetcher could not be turned into an arbitrary read primitive [120]. However, these CPUs had not been released when our research started, and we started investigating CPUs that were widely used at the time. We thus selected the latest generation of Intel CPUs released at that point.

> **Takeaway:** Prefetchers are both undocumented and susceptible to violating security guarantees, making them an attractive target for reverse engineering research. We target what was, at the time we started this Ph.D, the latest iteration of the most widely-used CPUs.

## 4.2. The Flush+Flush approach

### 4.2.1. Prefetcher quantum physics: the observer affects the experiment results

Prefetchers are trained by memory accesses, and their operation results in cache state differences. However, the primary measurement primitive for cache state is the timing of memory loads, which would also train the prefetcher. Consequently, this is an instance of experiments being influenced by the measurement, which may remind readers of a problem faced by quantum physicists. Some approaches can attempt to minimize the influence of the loads [19, 39, 94, 125], for instance, making a single memory access before resetting the experience and repeating it as many times as needed to measure every address. Unfortunately, it is impossible to eliminate the impact of loads entirely in this method, and this approach relies on the reproducibility of experiments, which is hard to check in this situation. This problem thus calls for a different method of cache state measurement.

> **Takeaway:** The conventional load-based method of measuring cache state interferes with prefetcher operation. As a result, a better measurement primitive would be suitable.

### 4.2.2. Rational for Flush+Flush

As shown by Gruss et al. [36], the `clflush` instruction exhibits some timing differences depending on the cache state. As this instruction removes a line from the cache and is pretty rare, it seems a reasonable hypothesis that this instruction does not trigger prefetcher operation and state update while being able to measure a precise cache line. This is the only method to measure cache state that does not use memory accesses.

Consequently, Flush+Flush seems a promising primitive to use should the hypothesis that it does not affect prefetcher operation be verified.

> **Takeaway:** Flush+Flush appears to be a cache state measurement primitive that could avoid the previous problem of interference of the measurement with the experiment.

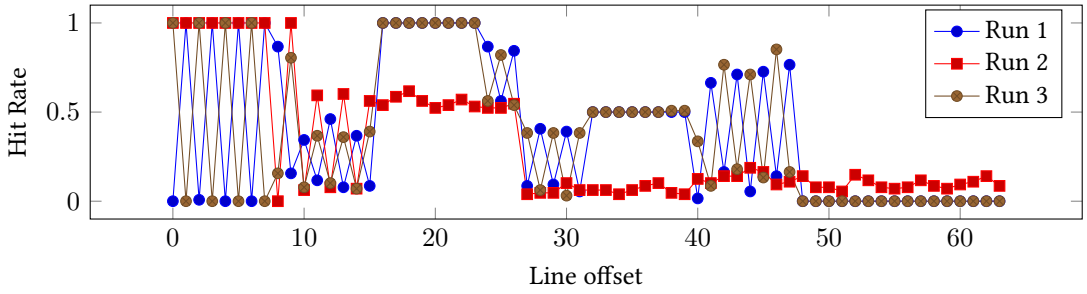



Figure 4.1.: Stream prefetcher enabled, hit rate (out of 128) within a page after accessing lines 0 to 9. The three runs show no apparent reproducibility and exhibit quite a few false positives and negatives.

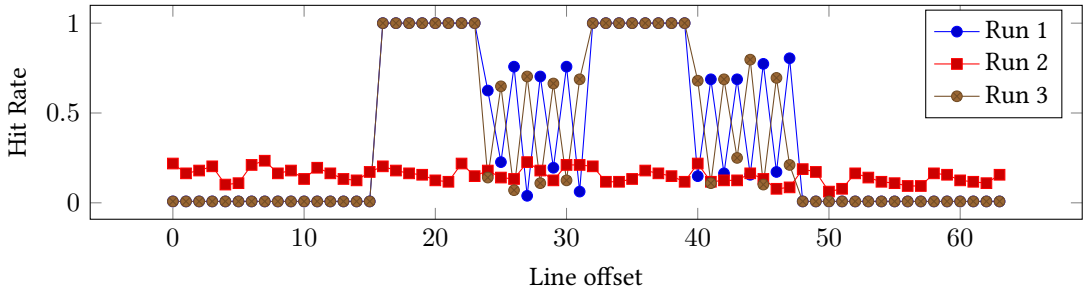



Figure 4.2.: Stream prefetcher enabled, hit rate (out of 128) within a page with no previous access. The three runs show no apparent reproducibility and exhibit many hits where they should not.

## 4.3. Preliminary experiments and unexpected results

Given the above, we started our study by attempting to observe the prefetcher result using Flush+Flush.

### 4.3.1. Naive Flush+Flush results

We first attempted to observe prefetcher activity using the code[1] from the original Flush+Flush paper [36]. This first study was initially done on a Sandy Bridge CPU. It was later reproduced on a more recent and relevant Coffee Lake CPU. In this section, we present an example of the surprising results we obtained, reproduced on the Coffee Lake machine[2] that was used for most of this thesis. Figures 4.1 to 4.4 show the graphs of hit rates for two different patterns, with each experiment being run two or three times. Even with no prefetchers, we can observe that some addresses exhibit no hits when accessed, and some un-accessed addresses exhibit a 100% hit rate.

In addition, the experiments exhibit a striking lack of reproducibility. This happens with

[1] `https://github.com/IAIK/flush_flush`
[2] The detailed configuration is shown in Section 4.5

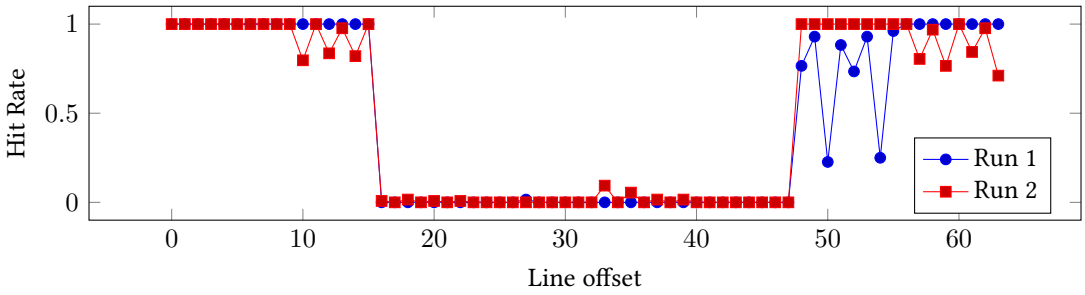



Figure 4.3.: No prefetcher enabled, hit rate (out of 128) within a page after accessing lines 0 to 9. The two runs show no apparent reproducibility and exhibit quite a few false positives and negatives.

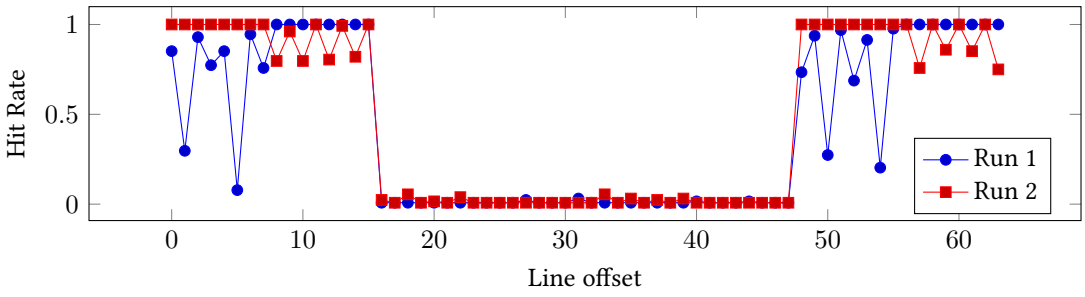



Figure 4.4.: No prefetcher enabled, hit rate (out of 128) within a page with no previous access. The two runs show no apparent reproducibility and exhibit many hits where they should not

dynamic voltage and scaling disabled. Thus another source of variation of `clflush` execution time exists and should be investigated.

> **Takeaway:** Monitoring a memory range using the Flush+Flush original code does not lead to reproducible results on the CPUs we study. There are both instances of 100% false positives and false negatives.

### 4.3.2. Bare-metal attempts

Attempting to eliminate the noise in the previous results obtained on a machine running Linux, we developed a framework that would allow us to run these experiments bare-metal, with no interference from the operating system. Among others, it aimed to ensure no interrupts occurred during the experiments and to get control of the virtual-to-physical address translation. This was the origin of the *dendrobates*[3] framework.

The results observed were quite similar; however, a given physical address would display reproducible results, whereas the previous experiments would see its result change significantly each time the physical addresses used to calibrate the execution time changed.

We measured for each core the median hit and miss time for every address in a 2 MiB huge page on our Coffee Lake machine. Figure 4.5 shows the result of this experiment for the first 128 cache lines. The full result of 4096 lines behaves similarly and does not provide further insight. 128 lines cover two 4 KiB pages.

Thus, we determined that these variations were not caused by operating system noise and were tied to physical addresses. These unexpected variations of timings explained why Flush+Flush did not behave as expected. The pattern in the variations, especially once we ran the same experiment for all cores, suggested it involved the sliced structure of the last-level cache used in these CPUs.

> **Takeaway:** We attempted a study of prefetchers on Sandy Bridge and then on Coffee Lake using the original Flush+Flush primitive and obtained surprising results. Upon further investigations on bare metal, it appeared the Flush+Flush primitive itself was not behaving as we expected, with significant variations in the typical `clflush` execution time for hit and misses alike depending on physical addresses, possibly owing to the sliced cache structure.

Following this unexpected result, we investigated the Flush+Flush primitive and built upon the work done in the original paper. This led to our first paper, *Calibration done right: Noiseless Flush+Flush*, published at DIMVA '21 and presented in Chapter 5, which studied in detail the impact of the sliced cache structure and interconnect on `clflush` execution time. Following this work, we returned to prefetchers in *Characterizing Prefetchers using CacheObserver*, accepted at SBAC-PAD 2022 and reproduced in Chapter 6. Finally, we also present the final Dendrobates framework in Chapter 7, including the development from this initial research and both papers.

[3] or more accurately *Dendrobates Tinctorius Azureus*, named after a blue frog, as it was aimed at dissecting CPUs from a company with a blue logo, Intel

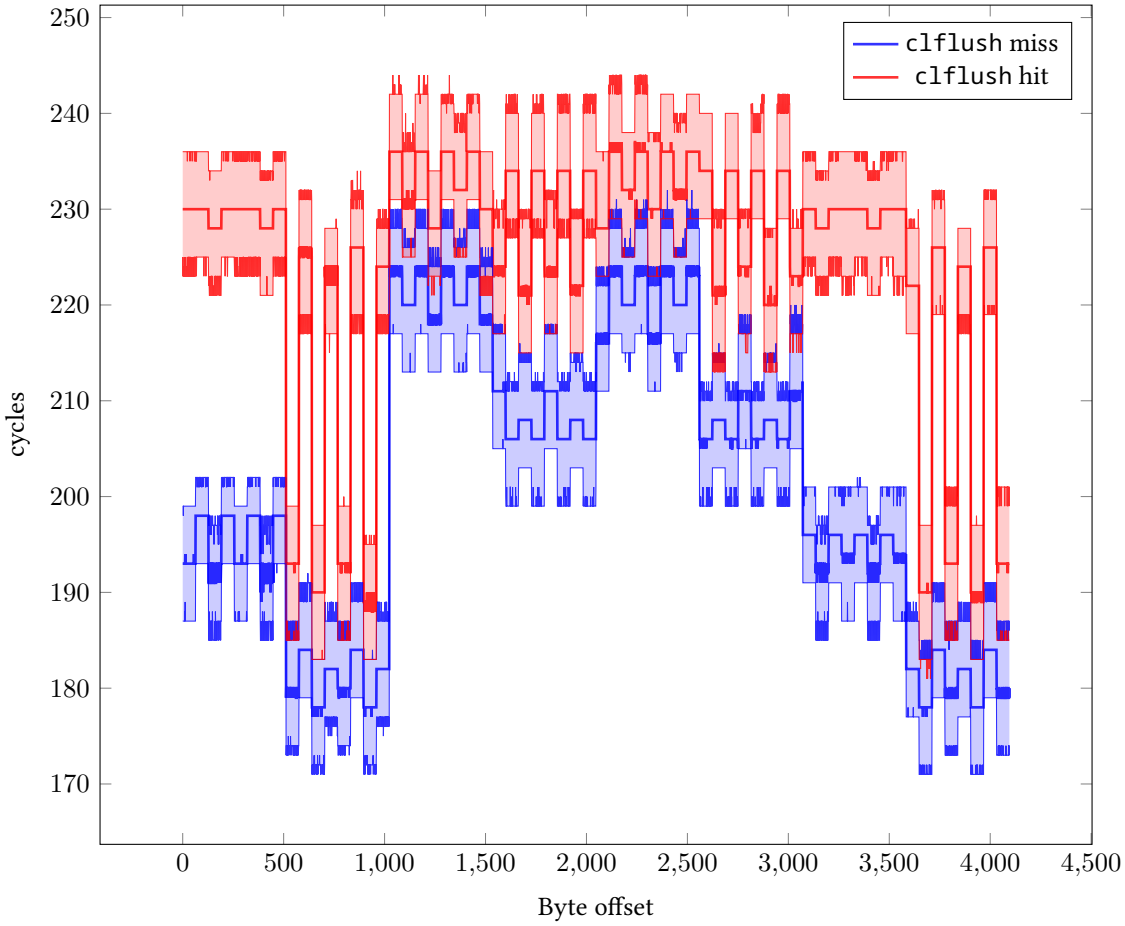



Figure 4.5.: Minimum, maximum, and median `clflush` hit (red) and miss (blue) time for every address in a 4 KiB physical page, on CFL. This figure shows that the execution times vary depending on the cache line accessed. The global distribution of execution times has significant overlap, but the overlap is much smaller for a given cache line.

## 4.4. The Dendrobates framework goals

Going back to the *dendrobates* framework, its initial goal was to enable microarchitectural study on bare metal.

It was designed with the following initial constraints and objectives in mind.

- *No preemption mode*: Unlike modern operating systems where interrupts are used to handle interaction with the various peripherals and the outside world and resource sharing, we want a system where experiments can run undisturbed and uninterrupted.
- *Control over the physical addresses*: The ability to control the virtual-to-physical mapping is important given that the physical page number bits are used as the input of the slicing functions, for instance.
- *Abstraction of the cache channel primitives*: To easily try out different primitives and approaches or compare different primitives, abstraction is an important part of the framework.
- *Zero-cost abstractions*: Microarchitectural experimentation is sensitive to overheads in the generated assembly code. Consequently, it is important to have a language where the experimental routine can be written without incurring overhead introduced by unneeded abstractions.
- *Ability to transmit information without an operating system*: We cannot rely on existing drivers for a bare-metal set-up. Hence we need to roll our own drivers to communicate with the outside world.

The initial goal was to *precisely measure the various sources of variation of* `clflush` *timing and determine the reliability of the Flush+Flush primitive.*

Given those constraints, we built the framework using Rust, a modern language with zero-cost abstractions, which had an existing ecosystem of libraries that made booting x86 CPUs in 64-bit mode straightforward. We wrote our own serial port driver that would use polling. Polling is generally frowned upon for its inefficiency compared to interrupt-driven drivers; however, in our case, we needed the driver to operate without relying on interrupts. The bare-metal mode ends up having control of the virtual-to-physical mapping.

On the one hand, Rust's more advanced features, including its generics system, allowed us to build various layers of abstraction related to cache channels. On the other hand, the language can access low-level features through intrinsics or even dropping to assembly.

One limitation of the bare-metal mode was that we only booted a single core. We supported several cores on a hosted mode that would run on top of a Linux distribution. After the first experiments showed the noise observed in hosted mode was identical to the bare-metal noise, we shifted back to running most of our research on the hosted mode.

Further development throughout the work added a significant amount of functionalities. These are generally written for the hosted mode. They include utilities to benchmark cache channels, an automated implementation of the calibration algorithm developed in Chapter 5, and a system designed to execute sequences of memory accesses while monitoring their impact on the cache, built as part of Chapter 6. The code released for each paper is an extract of the relevant part of the *dendrobates* framework, released alongside this thesis.

We detail in Chapter 7 the details of the architecture of the final framework.

Table 4.1.: Experimental configuration used in this thesis

| Shorthand | Wkl | Cfl |
|---|---|---|
| Computer type | Laptop | Workstation |
| Computer model | Dell Latitude 7400 | Dell Precision 3630 |
| Name* | Citron Vert | Cyber Cobaye |
| Operating system | Fedora 30 | Ubuntu 18.04.5 LTS |
| CPU | Intel Core i5-8365U | Intel Core i9-9900 |
| Microarchitecture | Whiskey Lake | Coffee Lake |
| Physical Cores | 4 | 8 |
| Threads | 8 | 16 |
| Base Frequency | 1.60 GHz | 3.10 GHz |
| Max Frequency | 4.10 GHz | 5.00 GHz |
| L1D size | 32 KiB per core | 32 KiB per core |
| L1D associativity | 8 ways | 8 ways |
| L2 size | 256 KiB per core | 256 KiB per core |
| L2 associativity | 4 ways | 4 ways |
| L3 size | 6 MiB ($4 \times 1.5$ MiB) | 16 MiB ($8 \times 2$ MiB) |
| L3 associativity | 12 ways | 16 ways |

*: This name may appear in the naming convention of the result files.

## 4.5. Experimental set-up

Most of the work in this thesis was done on a pair of machines near the end of the Sandy Bridge lineage, which are incidentally the laptop and the workstation pictured in Fig. 1.1. Table 4.1 presents those two configurations. Furthermore, a few additional experiments were run on Grid'5000 [9], with the precise details indicated where relevant.

Both machines present Dynamic Voltage and Frequency Scaling (DVFS), notably with an Intel feature named Turbo Boost. Turbo Boost allows the CPU to temporarily run a core or a few cores at a frequency that is not sustainable thermally in the long run, as long as the temperatures throughout the chip are low enough. However, extended workloads generally lead the CPU to throttle once it reaches the thermal limits.

As frequency variation is a significant impediment to our research work, we disable these variations by first disabling Turbo Boost, which boosts the frequency above a thermally sustainable one. We then need to disable frequency variations under this stable base frequency for energy-saving purposes. Under a Linux environment, this is done with the `cpufreq` driver and `cpupower` utility by selecting the `performance` frequency governor.

> **Takeaway:** Two machines were used, one equipped with an 8-core Coffee Lake CPU, referred to as CFL in the rest of the work, and one equipped with a 4-core Whiskey Lake CPU referred to as WKL.

With these first results in hand showing different timing behaviors for `clflush` depending on the physical addresses, we investigate the source of these variations in the next chapter.

# 5. Calibration done Right

Faced with the Flush+Flush unreliability results presented in the previous chapter, we investigated the `clflush` instruction further to understand its timing. This chapter presents this joint work with Clémentine Maurice that appeared at DIMVA 2021 as *Calibration Done Right: Noiseless Flush+Flush Attacks* [26].

In this chapter, we present the following contributions[1]:

1. We present a method to uncover the interconnect topology of Intel CPUs and apply it to Coffee Lake CPUs. We explain the variation of the execution time of `clflush` caused by topology on single-socket systems.
2. We measure the resulting error rate depending on the location of the attacker, victim, and cache slice accessed on single-socket machines and analyze the differences with dual-socket machines.
3. We benchmark the ideal capacity of the resulting improved covert channel, compared with Flush+Reload and a naive implementation of Flush+Flush. We show how these improvements make Flush+Flush a reliable side-channel primitive, on par with Flush+Reload.

The chapter is organized as follows: Section 5.1 first recalls some of the background knowledge and summarizes the reasons for this study of the Flush+Flush primitive. Section 5.2 then restates our experimental configuration. We then investigate the interconnect topology in Section 5.3 before exploiting the interconnect topology to improve the error rate of Flush+Flush channels in Section 5.4. We then evaluate the improved Flush+Flush, both as a covert channel and as a side channel (Section 5.5), and finally summarize the chapter and discuss potential future work (Section 5.6).

## 5.1. Background reminders and Motivation

As we have seen in Section 3.1.2, Flush+Flush is a stealthy and fast cache attack primitive that uses the timing of the `clflush` instruction depending on whether a line is cached. It has the potential of monitoring several addresses with less interference than Flush+Reload, as flushing does not trigger the prefetcher. However, we have just seen in Section 4.3 that a naive implementation of this primitive was noisy and unreliable in our experimental setup.

The choice of the threshold to distinguish between a flush hit and a flush miss is crucial to avoid noise. When looking at the timings for a single address and on a single run, it appears that there is a good separation between the hits (slower) and the misses (faster) for a single-socket

[1] The corresponding code is available at `https://github.com/MIAOUS-group/calibration-done-right`

system. However, the exact threshold may change from one run to another, even with a fixed frequency. The threshold also differs for different addresses.

We hypothesize that the variability is due to the complex topology of sliced caches, discussed in Section 1.5.2 and that accounting for these sources of variability significantly improves the quality of the channel, especially as the number of cores grows. Our experiments show that ignoring CPU topology can result in abysmal error rates, e.g., in some cases, a 45% error rate for a covert channel using a naive method for choosing the threshold. In the remainder of the chapter, we show that taking into account the topology and slices to compute tailored thresholds allows us to build a side channel with an error rate well under 0.01%. Therefore, Flush+Flush is not a noisy attack when crafted carefully, contrary to what was thought before.

## 5.2. Experimental setup

We run experiments on the two single socket systems described previously, the 8-core Cfl and the 4-core Wkl. As part of this, we had to reverse engineer the hash functions used to slice the last level caches of the 8-core Cflmachine, which we present in Appendix A.

We enable hyper-threading but disable turbo boost on those machines and set the intel_pstate driver to performance mode on all cores to stabilize the core frequencies. Additionally, we write a non-null value in each page before use, which prevents any optimization and involuntary page sharing involving the zero free page. The Zero free page on demand (ZFOD) optimization could otherwise come into play and introduce unexpected page sharing. While this could be investigated as a way to build covert channels between processes that do not share memory, this is not our goal here.

Some additional experiments were run on the Grid'5000 test-bed [9]; we will present a figure showing results on the 2x Intel Xeon E5-2630 v3 machine belonging to the testbed's `graoully` cluster.

## 5.3. Topology Modeling

In this section, we investigate the factors that influence the execution time of `clflush` to improve the Flush+Flush attack and propose a mathematical model with an associated ring topology. The only information we have from the Intel documentation is that the interconnect is a "bidirectional ring".

A `clflush` miss occurs when a cache line is not validly cached, which corresponds to a line in the I state. A line that has just been flushed is in the I state — the cache may have an entry in the I state or no entry at all, but it is equivalent at the cache coherency protocol level. A `clflush` hit occurs when the line is in any of the valid states. However, in practice, in a Flush+Flush attack, the cache line of interest transitions from an I to an E state when the victim core loads the line that has just been flushed. Therefore, the two relevant execution times are that of `clflush` on lines in the E state for hits and that of `clflush` on lines in the I state for misses. We study these timings depending on three parameters:

1. $A$: the attacker core that executes `clflush` on an address,

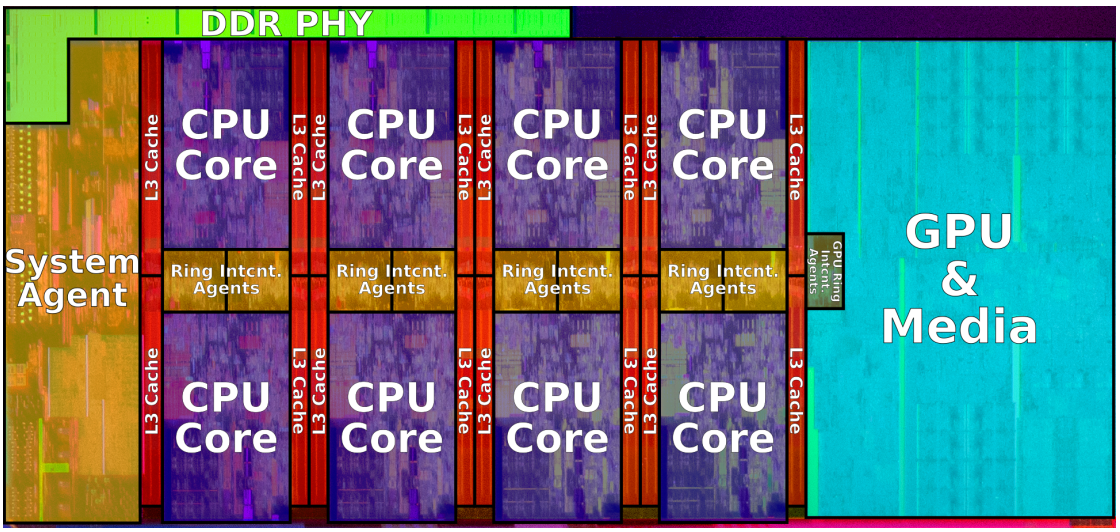



Figure 5.1.: Intel Coffee Lake 8-core die shot.
Image by Intel, annotated by WikiChip [21].

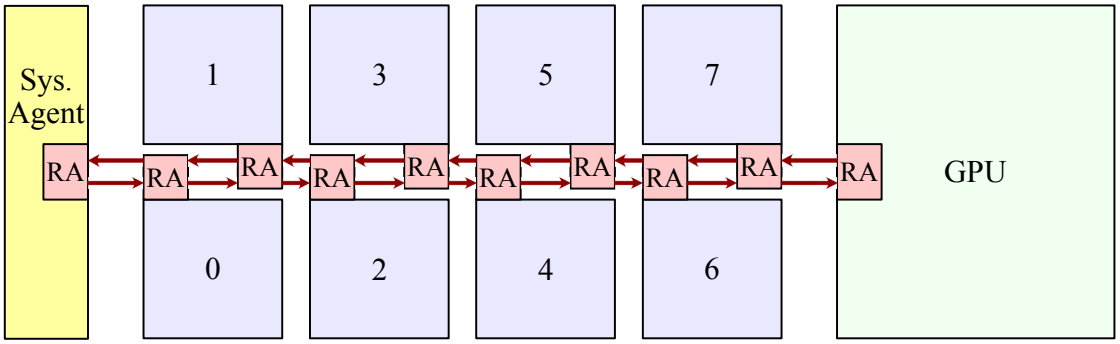
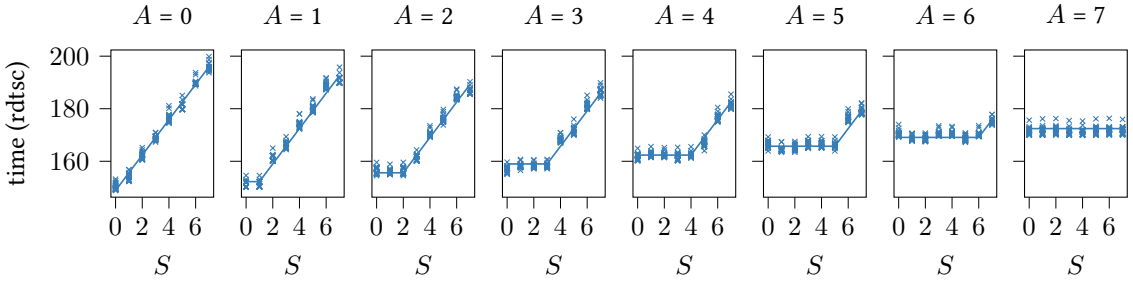
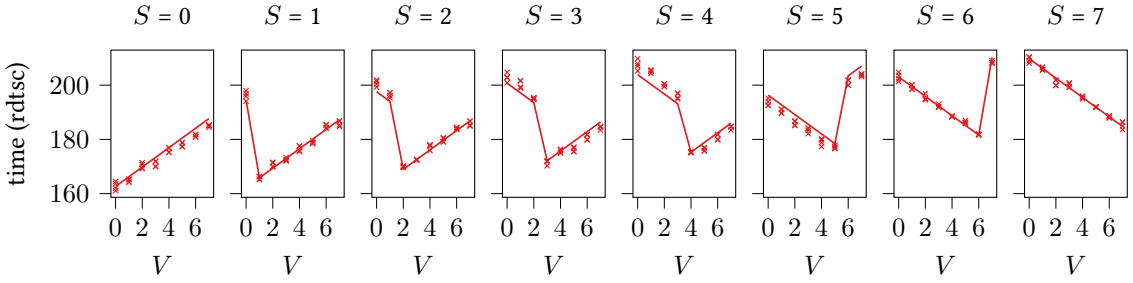



Figure 5.2.: Proposed i9-9900/8-core topology.
Each core (0 to 7, blue) has a ring agent (RA, red), handling memory accesses to the local L3 slice and communication over the ring. In addition, there are also ring agents for the GPU and the system agent that handles outside world communication, such as memory.




(a) For a cache line in the I state, depending on its slice $S$ for each attacker core $A$. There are 32 points per slice $S$, as we made one measurement for each attacker hyper-thread (2) and each victim logic core (16). Victim logic core has no impact on a miss.




(b) For a cache line in the E state, depending on the victim core $V$ for each slice $S$, using a fixed attacker core ($A = 0$). There are 4 points per victim core $V$, one per combination of attacker and victim hyper-threads.

Figure 5.3.: Median timings of `clflush` on the 8-core machine depending on the victim core $V$, the slice $S$, and the attacker $A$, along with the fitted model according to our proposed topology, which corresponds to our measurements.

2. $V$: the victim core that accesses the address and caches it in its L1 or L2,
3. $S$: the core that contains the last-level cache slice to which this address maps.

$V$ does not contribute to `clflush` miss timing as invalid lines are not cached in any L1/L2.

> **Takeaway:** We introduce the Attacker ($A$), Victim ($V$), and Slice ($S$) parameters that influence `clflush` execution time, and show the variations of `clflush` execution time depending on them.

### 5.3.1. Measurements and topology

For each attacker core, Figure 5.3a shows the time it takes to execute a `clflush` instruction on a cache line in the I state, depending on the slice. The first finding is that all 8 cores have a distinct timing pattern, which implies that the ring has no symmetry.[2] For each attacker, we

[2]Unlike the figure in Intel documentation [46] and the figure by WikiChip [21].

notice that slices with a lower core number than the attacker all have the same timing, while for slices with a higher number, the time increases with the distance between the attacker and the slice. Such a pattern only makes sense if the nodes form a single line and if the flow of message fo from the attacker to the slice, then from the slice to the system agent, back to the slice, and finally to the attacker. Consequently, attackers observe increasing variability in `clflush` miss time the closer they are to the system agent.

Figure 5.3b shows the time it takes to execute a `clflush` instruction on a cache line in the E state. Here, we notice an asymmetry in the core, which can be explained if the slice always sends recall requests in the same direction without knowing in which core the line is cached. We omit the graphs for other $A$ as they only show a simple linear offset depending on $|A - S|$.

Based on the topology description as a ring, the die shot in Figure 5.1 and our results, we propose the topology in Figure 5.2, with 8 cores aligned in a linear graph with forward and backward links. For a 4-core machine, similar measurements lead to a similar topology with only cores 0-3.

> **Takeaway:** The above measurements suggest the interconnect has the topology proposed in Fig. 5.2.

### 5.3.2. Mathematical model

The above timing measurements can be interpreted within the proposed topology, leading to a mathematical model that can be fitted and compared with the measurements. A misses results in a request to be sent on the ring from the core requesting the flush to the slice, which then sends a message to the memory, and then answers the same path in reverse, using each time the shortest route. The eviction time in state I, $t_I(A, S)$ is thus:

$$t_I(A, S) = C + h \times |A - S| + h \times |S - M|,$$

in which:

- $C$ is a constant base timing,
- $h$ is a constant corresponding to the time a round-trip hop on the ring takes,
- $M$ corresponds to the system agent location, $-1$.

Upon receipt of a request to flush a line in the E state, the slice sends a single message along the ring in one privileged direction. For core numbered from 0 to $\frac{n_{core}}{2}$ included, this is towards the higher numbered cores (and the GPU); otherwise, the message is sent towards the lower numbered cores. This message is passed around the ring until it reaches the victim core $V$ that has the line cached in its lower level cache (L1/L2). If the core is not in the initial direction, the message will follow the ring back in the other direction until it reaches the victim core. After discarding the necessarily clean line, the victim core sends a reply along the shortest path to

the slice. The eviction time in state E, $t_E(A, S, V)$ is thus:

$$t_E(A, V, S) = \begin{cases} C' + h \times |A - S| + h \times |R - (V - M)| & \text{if } S \leqslant \frac{N}{2} \text{ and } V < S \\ C' + h \times |A - S| + h \times |S - V| & \text{if } S \leqslant \frac{N}{2} \text{ and } V \geqslant S \\ C' + h \times |A - S| + h \times |S - V| & \text{if } S > \frac{N}{2} \text{ and } V \leqslant S \\ C' + h \times |A - S| + h \times |M - V| & \text{if } S > \frac{N}{2} \text{ and } V > S, \end{cases}$$

where:

- $C'$ is a different base time constant,
- $h$ is a constant, roughly how long a round-trip hop on the ring takes,
- $N$ is the number of cores,
- $R$ is the ring diameter in hops, corresponding to how many hops there are between the system agent and the GPU, and thus, in our case, $R = N + 1$.

In addition to our measurements, Figure 5.3a and Figure 5.3b present the fitted model for the 8-core machine, which appears to explain the behavior consistently.

> **Takeaway:** We have built a mathematical model of the behavior expected on the proposed topology and shown it was compatible with the experimental results.

**Summary** We have uncovered that while CPUs appeared to be arranged symmetrically in Intel's bidirectional ring, they are, in fact, aligned one after the other in a linear graph, with the system agent at one end and the GPU at the other. First, the `clflush` instruction timing is always influenced by the distance between the core requesting the flush and the slice where the address lives in the last-level cache. Second, in the I state, the timing will depend on the distance between the slice and the system agent, whereas in the E state, it will depend on how long a message sent along the ring will need to reach the core that currently has the line, and then go back to the slice. These findings are consistent with those by Paccagnella et al. [85].

> **Takeaway:** In this section, we have proposed a topology for the interconnect of the Coffee and Whiskey Lake CPUs, which should apply to CPUs as far back as Sandy bridge, and in agreement with concurrent work.

## 5.4. Improving error rate accounting for topology

### 5.4.1. Attacker models

We first define different attacker models depending on attacker capabilities. Then, we measure the error rate that can be achieved for each triple consisting of an attacker core, a victim core, and a slice. Finally, we also compute the average over all triples.

The attacker core can be set using the `sched_set_affinity` Linux system call. We, therefore, assume that the attacker always chooses the core with the lowest error rate. In some cases, the attacker may also control the victim's core, e.g., if she launches the process. The victim core can

always be found using the `/proc/pid` file system that gives the core affinity and the last core used.

The slice can be found using the physical address, but this information is usually unavailable to an unprivileged attacker. However, when the hash function is linear, it is possible to define an equivalence class of addresses within a page that all belong to the same slice. It is not possible to know which equivalence class corresponds to which physical slice *a priori*, but the pair made of the page and the result of the hash function defines an equivalence class of virtual address with the same timing impact. We name this equivalence class $\tilde{S}$. Using timing measurements, within each page, each equivalence class can be, *a posteriori*, attributed to a precise physical slice, but we do not use this attribution for our attacks.

If the attacker launches a covert channel, she can pick the addresses used to communicate and, therefore, the optimal equivalence class. In a side-channel attack, the attacker cannot pick the addresses to monitor but usually knows the equivalence class, as she knows both addresses and hash functions. We still present models where the attacker ignores the slices to compare the previous naive models with the ones that yield the best attacks.

- *Global Threshold (GT)*: The simplest model, using a single threshold that minimizes the average error rate over all *(attacker, victim, slice)* triples. This is a topology oblivious attacker, as in the initial Flush+Flush attack [36].
- *Best $A$, Known $V$*: The attacker knows on which core the victim is running and chooses the attacker core it runs on. The attacker computes a single threshold for all addresses, ignoring the impact of cache slices.
- *Best $AV$*: The attacker can pick the cores both the victim and the attacker are running on, e.g., in the case of a covert channel or a side-channel attack in which the attacker launches the victim process. It ignores the impact of slices.
- *Known $\tilde{S}$*: The attacker does not know on which core she or her victim runs but takes the slices into account, using per-slice thresholds. We use this model for comparison with the GT model.
- *Best $A$, Known $\tilde{S}V$*: The attacker pins her process to the best core, knows the victim's core, and takes into account the slices. This is a realistic attacker model. To be compared with *Best $A$, Known $V$* model.
- *Best $AV$, Known $\tilde{S}$*: This is the most powerful side-channel attacker that can pin both the attacker and victim.
- *Best $AV\tilde{S}$*: This is the best covert channel attack model, where the attacker chooses the cores and an address in a slice that yields the best results.

> **Takeaway:** There are several possible models of attackers when running cache attacks and cache covert channels, depending on whether the attacker can know or even control the values of $A$, $V$, and $S$.

Table 5.1.: Results for each attacker model on the 4-core and 8-core machines. U. means Unknown and K. Known.

| | 4-core machine | | | | 8-core machine | | | |
|---|---|---|---|---|---|---|---|---|
| | Error rate | $A$ | $V$ | $\tilde{S}$ | Error rate | $A$ | $V$ | $\tilde{S}$ |
| GT | 14.0% | U. | U. | U. | 25.1% | U. | U. | U. |
| Best $A$, Known $V$ | 6.07% | 3 | K. | U. | 10.5% | 7 | K. | U. |
| Best $AV$ | 0.176% | 7 | 0 | U. | 0.115% | 7 | 8 | U. |
| Known $\tilde{S}$ | 11.6% | U. | U. | K. | 22.8% | U. | U. | K. |
| Best $A$, Known $\tilde{S}V$ | 3.16% | 5 | K. | K. | 7.18% | 7 | K. | K. |
| Best $AV$, Known $\tilde{S}$ | 0.103% | 7 | 0 | K. | 0.0174% | 1 | 0 | K. |
| Best $AV\tilde{S}$ | $4.96 \times 10^{-3}$% | 3 | 3 | 3 | 0 ($< 2^{-20}$) | 2 | 7 | 14 |

### 5.4.2. Experimental results on the error rates

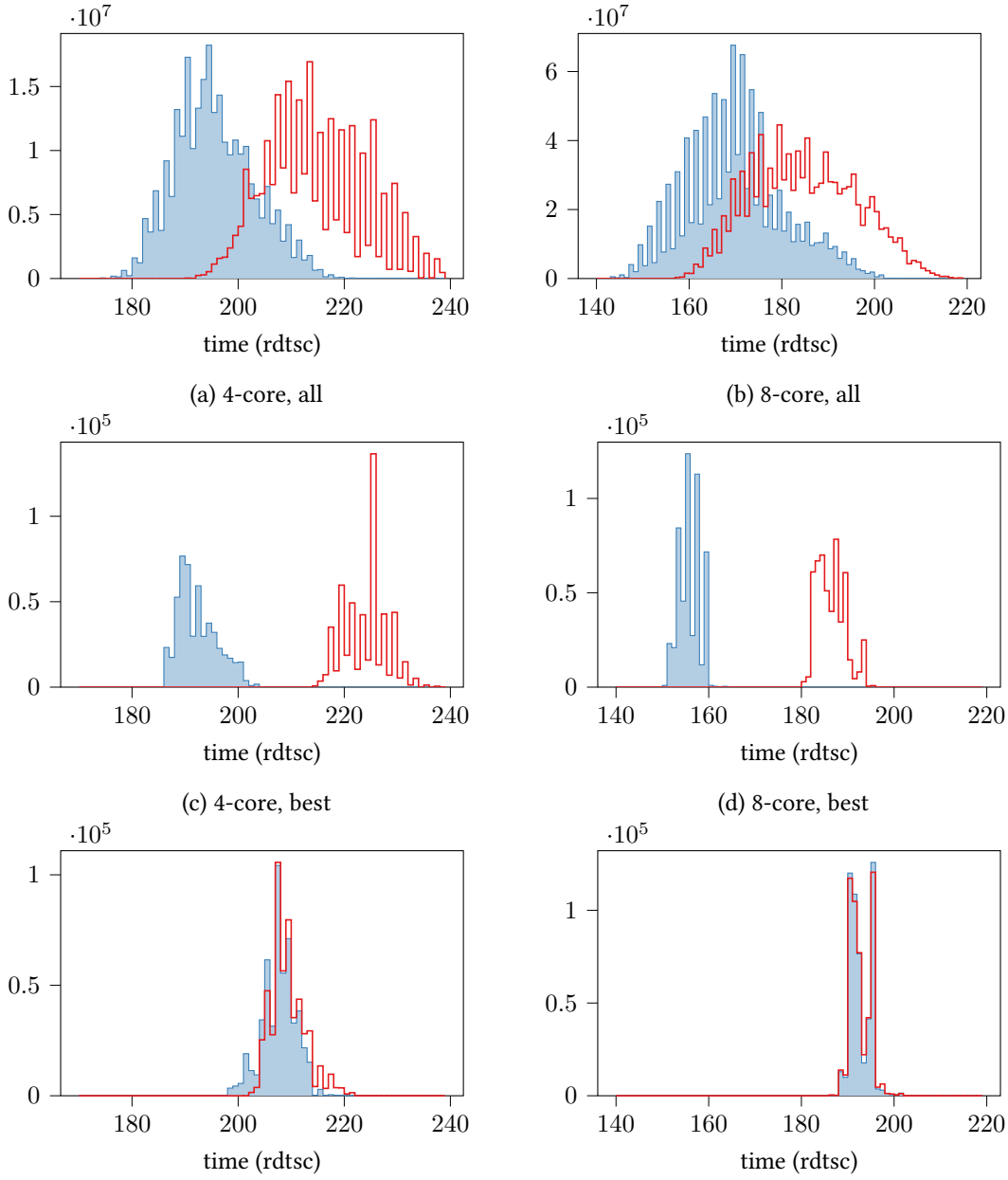
For each ($A$,$V$,$\tilde{S}$) we make $2^{20}$ measurements, $2^{19}$ hits (in E state), and $2^{19}$ misses (in I state). We time how long `clflush` takes to execute in each case using the `rdtsc` instruction and build a histogram of the execution time distribution. From these histograms, we can evaluate the number of hits and misses that would be correctly or incorrectly classified using a threshold and determine thresholds that minimize the average error rate for each model, along with the corresponding average error rate. We present three such histograms above:

- Figures 5.4a and 5.4b: the histograms for all attackers, victims, and slices.
- Figures 5.4c and 5.4d: the histograms on the best choice of the attacker, victim, and slice equivalence class in the *Best* $AV\tilde{S}$ attacker model.
- Figures 5.4e and 5.4f: the histograms on the most unfavorable choice of the attacker, victim, and slice, with severe overlap between the two distributions.

Table 5.1 shows the results for the 4-core and 8-core machines, indicating for $A$, $V$ and $\tilde{S}$ whether they are unknown, known, or chosen in each case. For the 8-core machine, we observe a staggering difference between the 25% error rate of the *GT* attacker model, to the close to 0% error rate of the *Best* $AV\tilde{S}$ model (less than 1 error per $2^{20}$ measures).

**Summary** Choosing the attacker and victim locations significantly improves the accuracy over the unreliable global threshold. On top of that, using a per-slice threshold provides a further boost. However, when the victim cannot be chosen, accounting for slices gives a much more significant boost. Lastly, choosing the best combination of the attacker, victim, and slice gives close-to-perfect error rates.

> **Takeaway:** Accounting for the interconnect topology and using distinct thresholds depending on $A$, $V$, and $S$ provides significantly improved accuracy, reaching error rates well below 1%.




(a) 4-core, all

(b) 8-core, all

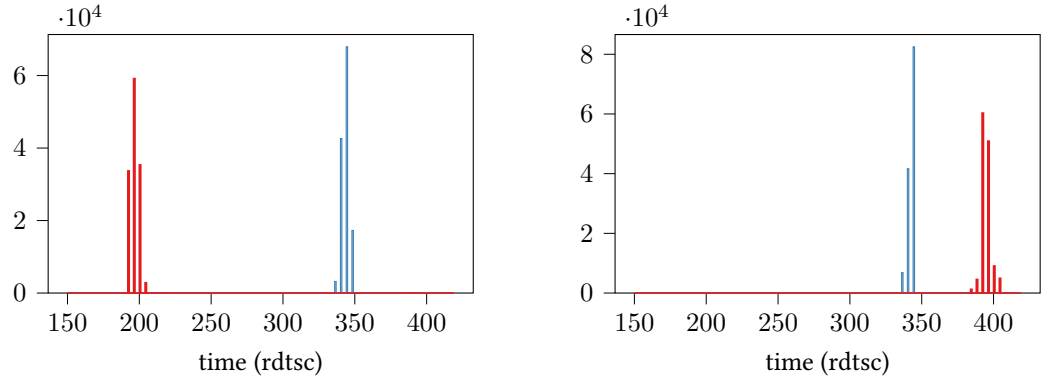
(c) 4-core, best

(d) 8-core, best

(e) 4-core, worst

(f) 8-core, worst

Figure 5.4.: Histograms for both machines of hit (outlined, red) and miss (filled, blue) `clflush` timing distributions for: – a, b: the superposition of all possible $(A, V, \tilde{S})$ triples (Average in the *GT* model). – c, d: the best possible $(A, V, \tilde{S})$ choice (*Best* $AV\tilde{S}$ model) – e, f: the worst possible $(A, V, \tilde{S})$ choice.




(a) $A$ and $V$ in the same socket.

(b) $A$ and $V$ on different sockets.

Figure 5.5.: Histograms of hit (red) and miss (blue, around 340) `clflush` timing distributions for two different $(A, V)$ pairs on a 2x Intel Xeon E5-2630 v3 machine.

### 5.4.3. The case of dual-socket machines

In dual-socket machines, there is no cache shared by the entire coherency domain. Coherence is maintained using bus snooping and ECC bits in the DRAM to store some coherency information [46, 75]. Thus, `clflush` behavior differs significantly from single-socket systems, depending on the attacker's and the victim's locations. The slice is not attached to a specific socket as each socket has its own last-level cache; thus, its contribution here was not studied in detail.

Figure 5.5a shows that when the victim is in the same socket as the attacker, we observe that a hit is faster than a miss. This makes sense if the socket last-level cache has the coherency info of the accessed line in the E state, whereas it needs to reach out to the DRAM directory on a miss. However, when the victim is located in the other socket, *a hit is slower than a miss*, as shown by Figure 5.5b. This can probably be explained because more communication is required in the former case to cause the remote core to evict and then update the DRAM directory.

Overall, a simple threshold model will give poor results if the sockets on which the attacker and victim reside are not controlled. However, a dual threshold model may give good quality results, distinguishing same-socket hits, misses, and remote-socket hits. A detailed model accounting for the attacker's and the victim's locations would also provide high-quality results.

> **Takeaway:** On dual-socket systems, `clflush` involves communication between the two sockets, with `clflush` of a line in the E state locally being faster than an invalid line, itself faster than flushing a line held exclusively on the other socket. `clflush` is still exploitable but requires topology awareness or a pair of thresholds.

## 5.5. Evaluation

This section evaluates our improved Flush+Flush primitive on a covert channel and a side-channel attack on the AES T-tables implementation.

Table 5.2.: Result of covert channel benchmarking

| | 4-core machine | | | 8-core machine | | |
|---|---|---|---|---|---|---|
| Channel | Capacity | Bit rate | Err. rate | Capacity | Bit rate | Err. rate |
| Naive F+F | 1.01 Mbit/s | 2.96 Mbit/s | 20% | 1.88 Mbit/s | 5.89 Mbit/s | 23% |
| Opt. F+F | 2.99 Mbit/s | 3.03 Mbit/s | 0.1% | 5.81 Mbit/s | 5.81 Mbit/s | 0.005% |
| F+R | 2.88 Mbit/s | 2.91 Mbit/s | 0.1% | 5.57 Mbit/s | 5.57 Mbit/s | 0.0005% |

### 5.5.1. Building a better channel

**Protocol** We implement a framework to benchmark covert channel ideal bandwidth with different primitives. We use the same protocol for each primitive. The benchmark uses two threads in the same process and an optimized synchronization primitive. Such an ideal synchronization is unlikely to exist in real-world implementation but allows us to measure theoretical limits of the channel itself. Real-world channels are likely to observe a lower bit rate and a corresponding decrease in true capacity but with similar error rates.

In practice, we use several shared pages; within each, we pick an address in the optimal $\tilde{S}$. We also synchronize on a per page basis indicating which thread can currently access the page (to transmit or receive), using mutable shared memory as the ideal synchronization primitive. Once done with a page, threads hand the page over to the other threads by flipping the per-page bit.

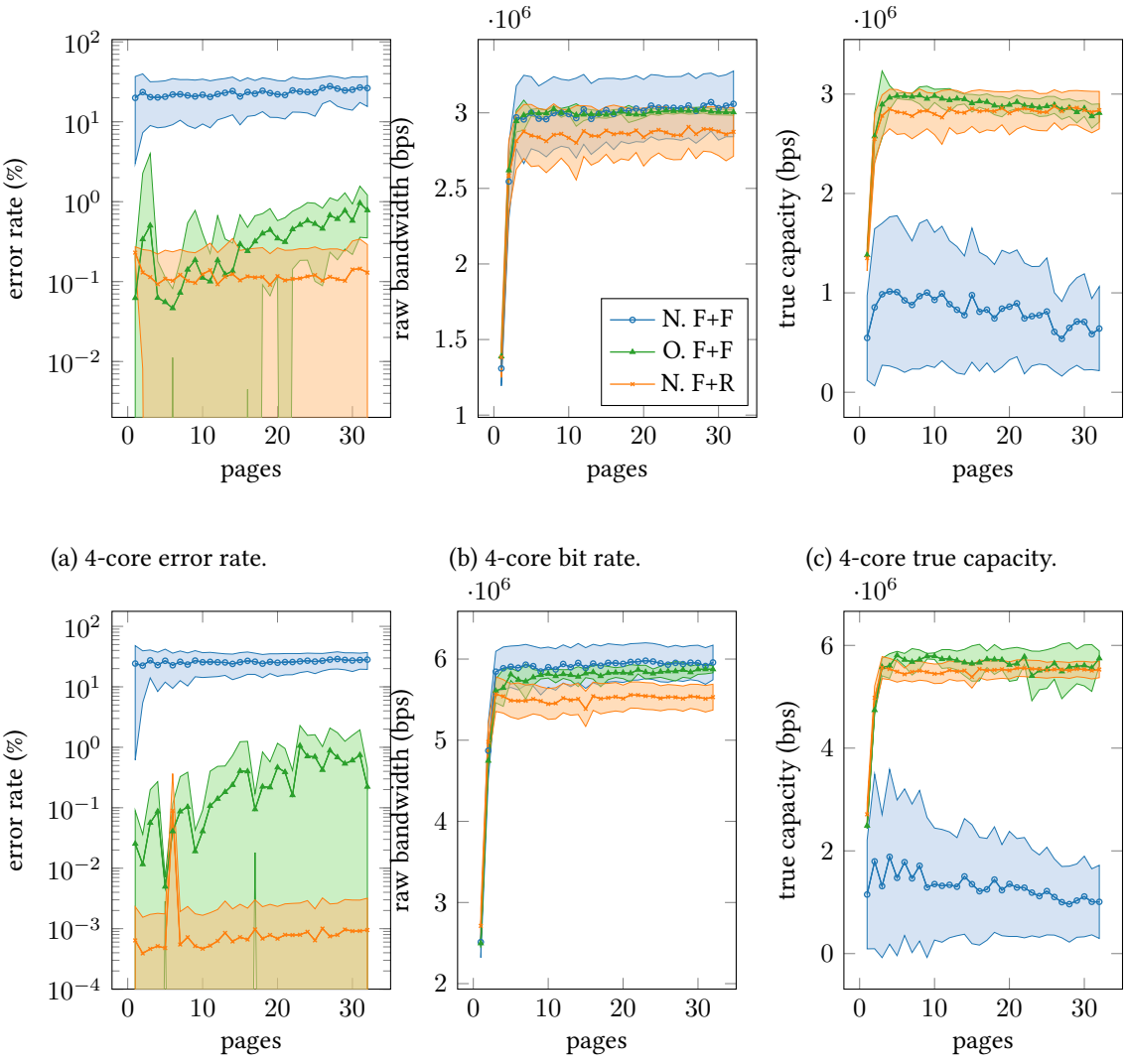
**Implementation** We implement three covert channels with different primitives: 1. a single threshold *naive* Flush+Flush, with no core pinning ($GT$ model), 2. a single threshold Flush+Reload that does not need to account for topology, and 3. a topology-aware Flush+Flush using the *Best* $AV\tilde{S}$ attacker model.

**Evaluation** For each channel and machine, we evaluate its raw bit rate $C$, error rate $p$, and true capacity $T = C \times (1 + p \log_2 p + (1-p) \log_2 (1-p))$ [81].

**Results** We run our experiments on both machines mentioned in Section 5.2. Figure 5.6 shows, for each machine, statistics on the performance of the covert channels depending on the number of pages used: the average error rate, the raw bit rate, and the true capacity of the resulting channel.

As shown by table Table 5.2, our carefully calibrated Flush+Flush yields a threefold increase in bandwidth on both machines compared to the naive Flush+Flush, and provides a bandwidth higher than Flush+Reload by 3 to 4 %. We conclude that Flush+Flush is now a compelling alternative to Flush+Reload.

> **Takeaway:** Our optimized Flush+Flush can achieve an error rate close to Flush+Reload and provides a channel with better true capacity, thanks to the higher bit rate.




(a) 4-core error rate.

(b) 4-core bit rate.

(c) 4-core true capacity.

(d) 8-core error rate.

(e) 8-core bit rate.

(f) 8-core true capacity.

Figure 5.6.: Covert channel performance depending on the number of pages used.

### 5.5.2. AES T-tables attack using Flush+Flush

**AES T-tables implementation** The AES T-tables implementation is well-known to be vulnerable to side-channel attacks; we, therefore, use it as a benchmark to compare our Flush+Flush implementation [1, 4, 5, 14, 37, 38, 84, 127]. We compare our improved Flush+Flush implementation with per-slice thresholds to the naive version of Flush+Flush and the Flush+Reload attack. We attack the OpenSSL 1.1.1g library, compiled with `no-asm` and `no-hw` to enable T-tables. For this experiment, prefetchers are enabled on both machines.

T-tables are an implementation of an AES round using lookups in tables. The lookup in the first round is $T_j[p_i \oplus k_i]$, where $0 \leq i \leq 16$ and $j$ is the remainder of $i$ divided by 4 (`j = i & 0x3`). With 4-byte elements and 64-byte cache lines, there are 16 entries per cache line, and a cache attack can only monitor the upper 4 bits of $p_i \oplus k_i$. See Osvik et al. [84] for a detailed explanation.

**Attacking the T-tables** We run the attack using all three side channels with the attacker and the victim in the same thread. For the naive Flush+Flush and Flush+Reload, the core is chosen randomly. For our improved Flush+Flush, we chose the best core according to Section 5.4, with the model *Best $AV$, Known $\tilde{S}$*.

We run the experiment with two different keys. The first is the null key, and the other is a key with $k_0 =$ `0x51`. In this chosen-plaintext attack, a byte of the plaintext is set to fixed values (`0x00`, `0x10`, `0x20`, by increments of `+0x10`), while the remainder is chosen randomly. In this case, one of the T-table's cache lines (depending on the fixed byte value) is deterministically accessed, while the others are not always accessed and have a higher number of misses. Such cache lines show distinctive patterns that identify a byte of the key when plotting the misses. Notably, the null-key pattern is diagonal.

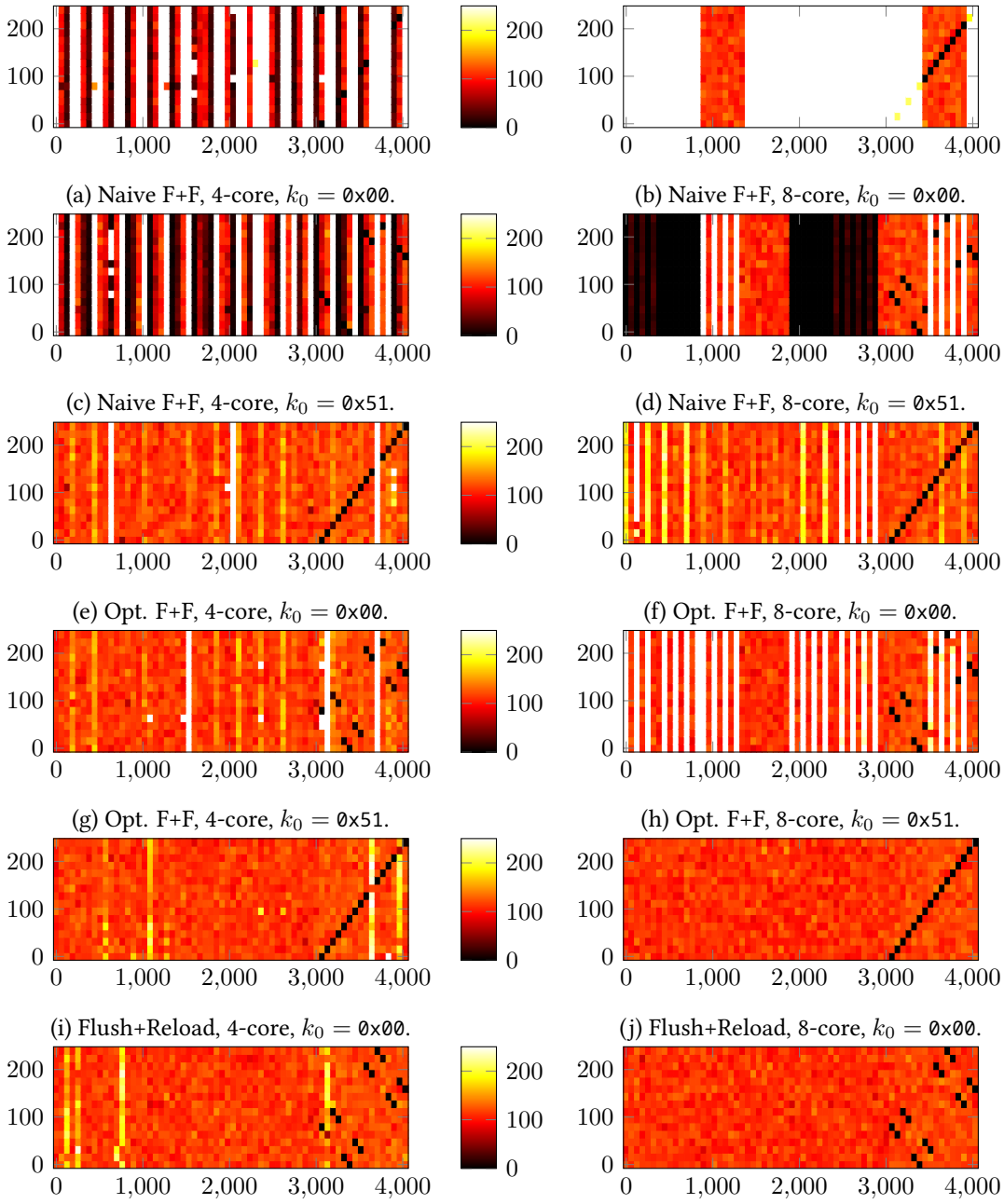
**Results** We observed that a naive Flush+Flush attack will show some lines with all hits or all misses, due to the threshold depending on the slice (see Figures 5.7a, 5.7b, 5.7c and 5.7d). Using a per-slice threshold (see Figures 5.7e, 5.7f, 5.7g and 5.7h) allows us to achieve an accuracy similar to Flush+Reload (see Figures 5.7i, 5.7j, 5.7k and 5.7l). Again, accounting for the contribution of slices and CPU interconnect to `clflush` timing variations makes an optimized Flush+Flush channel competitive with Flush+Reload and improves the reliability over naive Flush+Flush.

> **Takeaway:** Compared with the Naive implementation, our improved Flush+Flush provides far more exploitable results in a side channel setting and provides results close in quality to Flush+Reload, even though Flush+Reload remains more accurate.

## 5.6. Conclusion and perspectives

### 5.6.1. Summary

The interconnect topology of Intel CPU plays a more significant role than was previously known in cache attacks, particularly Flush+Flush. A naive Flush+Flush implementation that does not account for the topology yields poor error rates, especially as the number of cores increases.




(a) Naive F+F, 4-core, $k_0 =$ `0x00`.

(b) Naive F+F, 8-core, $k_0 =$ `0x00`.

(c) Naive F+F, 4-core, $k_0 =$ `0x51`.

(d) Naive F+F, 8-core, $k_0 =$ `0x51`.

(e) Opt. F+F, 4-core, $k_0 =$ `0x00`.

(f) Opt. F+F, 8-core, $k_0 =$ `0x00`.

(g) Opt. F+F, 4-core, $k_0 =$ `0x51`.

(h) Opt. F+F, 8-core, $k_0 =$ `0x51`.

(i) Flush+Reload, 4-core, $k_0 =$ `0x00`.

(j) Flush+Reload, 8-core, $k_0 =$ `0x00`.

(k) Flush+Reload, 4-core, $k_0 =$ `0x51`.

(l) Flush+Reload, 8-core, $k_0 =$ `0x51`.

Figure 5.7.: Results of the T-table attack using a Naive Flush+Flush, Optimized Flush+Flush, and Flush+Reload side channels. Each column represents an address, and each row corresponds to a different value of the first byte of the chosen plaintext, with the remaining bytes filled randomly. The color scale cuts off lines with too many misses; T-tables that are deterministically accessed have very few misses and reveal key bits.

We reverse engineer this topology and study its timing impact on the `clflush` instruction. Using these insights, we significantly enhance the Flush+Flush primitive by accounting for the topology during the calibration step. Consequently, we recommend taking into account these findings in the calibration step, measuring timings for all possible combinations of the attacker, victim, and home slice locations, and then determining the best thresholds depending on the attacker model. Therefore, our results demonstrate that the Flush+Flush primitive is as reliable as Flush+Reload, with further advantages in stealth and lesser susceptibility to prefetcher noise. Within the context of this thesis, it thus becomes a viable primitive for our prefetcher reverse engineering effort.

> **Takeaway:** Using a topology-aware calibration, Flush+Flush can be a reliable primitive suitable for prefetcher reverse engineering.

Within the wider scope of this thesis, we can answer **Q3** and **Q4**, as defined in Section 2 of the Introduction:

- **Q3**: *How can we explain and model the variations of* `clflush` *execution times?* The variations in `clflush` execution times are primarily caused by the communication on the interconnect, which depends on the cache line's coherence state and the location of the various cores involved.
- **Q4**: *How can Flush+Flush reliability be improved to build a channel as reliable as Flush+Reload?* By accounting for the topology of the sliced cache, it is possible to use appropriate thresholds for the various slices and locations of the cores involved. When feasible, one should select locations that reduce the overlap between the distributions of hit and miss times.

### 5.6.2. Perspectives

We have explored the timing of `clflush` for two coherence states, but using our framework, it should be possible to set up lines in other coherence states, such as shared (S) and modified (M). Those states do not impact side-channel research but can help better understand CPU memory hierarchy and performance.

> **Takeaway:** Cache attacks must be designed with the cache coherence protocol in mind.

The impact of frequency on timing channels, especially those relying on small differences, is significant. Most attacks are described at a steady frequency, but in a realistic setting, frequency scaling can severely hamper them. A model of instruction execution time depending on the frequency could mitigate this variability.

Intel large server CPUs starting with Skylake Scalable Processors (SP) no longer use inclusive caches. However, the ISA still requires that `clflush` flushes a cache line from the entire coherency domain. It should thus be possible to use the `clflush` instruction to attack such systems, an approach that [134] has not covered. These systems also use a different topology that warrants further inquiry.

`clflush` also behaves differently on multi-socket systems, as shown in Section 5.4.3, in a way that is not always tractable with a simple global threshold model. Further work could evaluate

the benefits of dual-threshold versus per $A, V, \tilde{S}$ threshold models and the performance of channels built in this way.

More generally, the Flush+Flush technique has not been studied on AMD CPUs, which implement the same `clflush` instruction. Yet, implementing a constant time `clflush` is probably more complex than the current non-constant time implementation. Consequently, unless manufacturers consider `clflush` non-constant timing an issue, the primitive will likely remain exploitable on future CPUs and may also be exploitable on AMD CPUs.

In addition, the ARM v8 ISA introduced its own cache flushing instruction, the `DC` data cache maintenance family of system instructions [7], which may be made available in user mode by the operating system. Consequently, evaluating whether this instruction is constant-time and surveying if any operating systems enable it would be valuable results.

# 6. L2 prefetchers in Intel Whiskey and Coffee Lake CPUs

With a reliable Flush+Flush in hand, built in the previous chapter, we go back to our initial objective of studying prefetchers. This chapter presents joint work with Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour, accepted at SBAC-PAD 2022 under the title *Characterizing Prefetchers using CacheObserver*.

This chapter presents our novel approach to prefetcher reverse engineering and its application to the L2 prefetchers in Intel CPUs. This chapter attempts to answer the following questions:

- (**Q6.1**) How can we build a detailed view of prefetch activity?
- (**Q6.2**) What is the impact of using load instructions for measurements on prefetch activity compared with `clflush`?
- (**Q6.3**) How does the Intel L2 Stream prefetcher behave, especially on the first few accesses in a page?
- (**Q6.4**) Do the various prefetchers in Intel CPUs interact?

We make the following contributions:

1. We characterize hardware prefetchers using `clflush`.
2. We develop CacheObserver, a framework to visualize prefetches resulting from memory access sequences.
3. We uncover various properties of the L2 Stream prefetcher on Intel Whiskey and Coffee Lake CPUs.
4. We show that the L2 Stream prefetcher and the L2 Adjacent Cache Line prefetcher interact.

The chapter is organized as follows: Section 6.1 recalls background information and related work. Section 6.2 then describes the CacheObserver framework. Section 6.3 specifies the setup of the experiments we ran. Section 6.4 presents the experimental results on the L2 Stream prefetcher, while Section 6.5 the results on the other L2 prefetcher and prefetcher interactions. Section 6.6 discusses the advantages and limitations of our approach and Section 6.7 sums up the results and future line of investigations.

## 6.1. Background reminders

We have seen in Chapter 1 how important caches were for performance and, in Chapter 2 how hardware prefetchers were used to overcome cold misses. Prefetchers can give manufacturers a

competitive edge and are mostly undocumented, even though they may have a security impact (Sections 3.1.3 and 3.2.2).

Prefetcher reverse engineering is thus a valuable contribution to the security field, in addition to being helpful to the communities of high-performance computing and academic prefetcher designers. However, this endeavor is hampered by the inability to directly observe the prefetcher activity, as caches themselves lie below the ISA. Consequently, microarchitectural channels or performance counters must be used to deduce information about the state of the cache and the prefetcher behavior (Section 3.1.1). As presented in Section 4.2, we selected an approach based on cache channels, using execution times. More precisely, we selected Flush+Flush as it would likely avoid the observer interference problem Flush+Reload is subject to. This interference arises from the fact that the prefetchers may be trained by load instructions, which are part of Flush+Reload measurements. This approach had been used by the various studies [19, 94, 105] which we summarized in Section 3.2.2, with more work having been done on L1 prefetchers than L2 ones.

Despite the attractiveness of Flush+Flush, obstacles arose that led to further investigation of `clflush` execution time, as presented in Section 4.3 and Chapter 5, resulting in an optimized topology-aware calibration algorithm and a reliable Flush+Flush primitive, which we will use as a foundation for the CacheObserver framework.

### 6.1.1. Prefetchers on modern Intel CPUs

Recall from Section 2.4.1 that since the Nehalem architecture (2008), Intel CPUs include four disclosed prefetchers that can be disabled independently by Model Specific Register (MSR) 420. This MSR is also documented on Sandy Bridge CPUs and still seems functional as of Coffee and Whiskey Lake (2018) CPUs; we reproduce here Intel's documentation in Table 6.1, identical to Table 2.1.

We are chiefly interested in the L2 Stream prefetcher. We have seen in Chapter 2 both the original concept by Jouppi [51, 52], defining a *stream* as a sequence of accesses to consecutive lines in increasing or decreasing order; and various academic and industry papers related to the implementation of such prefetchers [28–30, 44, 57, 88, 104, 111, 112, 115].

Prefetchers can be described with two parameters: their *distance*, *i.e.*, how far from the current access is the line fetched, and their *degree*, *i.e.*, how many lines they fetch upon a single memory access [94].

A first study of the L2 Stream prefetcher on Kaby Lake CPUs was attempted in [94]. They confirmed through reverse engineering many of the prefetcher's properties described in Intel's documentation [46]. Precisely, the prefetcher issues requests to consecutive cache lines in the positive (increasing addresses) or negative (decreasing addresses) direction. It is possible to start a stream and then trigger prefetches with an access further away in the same page.

They also estimated at 16 the number of pages the prefetcher tracks, a number consistent with Intel's documentation, which indicates that both a positive and a negative stream can be tracked per page, for a total of 32 streams. Furthermore, the prefetcher state is shared between hyper-threads, and they were thus able to build a covert channel over it. Lastly, they showed it does not operate across 4 KiB page limits, even with 2 MiB huge pages.

Table 6.1.: Prefetchers disclosed by Intel for Sandy Bridge CPUs, in [47] Vol. 4 p 2-179, repeated from Table 2.1 .

| Bit | Intel Description |
|---|---|
| 0 | L2 Hardware Prefetcher Disable (R/W)<br>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache. |
| 1 | L2 Adjacent Cache Line Prefetcher Disable (R/W)<br>If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes). |
| 2 | DCU Hardware Prefetcher Disable (R/W)<br>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache. |
| 3 | DCU IP Prefetcher Disable (R/W)<br>If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines. |

## 6.2. The CacheObserver framework

To reverse engineer prefetchers, we want to observe the state of the cache after a sequence of memory accesses. The cache state is a microarchitectural state and cannot be queried directly: it is usually measured by timing memory loads. However, such loads may trigger prefetches and interfere with the experiments [19, 94, 105]. Hence our choice of the Flush+Flush primitive, which times the `clflush` instruction.

Our CacheObserver[1] framework uses the code base[2] of Didier et al. [26], implementing reliable calibration of Flush+Flush channels, which are both are written in Rust. The core of the framework is the *Prober* object, which manages a set of pages and runs sequences of memory accesses called *patterns*, measuring their impact on each cache line of the page, designated as $p$ (probe).

> **Takeaway:** Our framework uses Flush+Flush to monitor prefetcher activity in reaction to an access pattern. It cycles through a set of page, executing the pattern once per page, and then measure the state of every cache line in the page.

### 6.2.1. Prober

Using a load instruction to measure cache state, we can only do one line per experiment run and must repeat it 64 times, once per offset in the page. However, if we assume `clflush` does not affect the prefetcher state, a single run of a pattern can give the state of all 64 cache lines.

[1] `https://anonymous.4open.science/r/CacheObserver-B3CD/`
[2] `https://github.com/MIAOUS-group/calibration-done-right`

```rust
pub unsafe fn rdtsc_fence() -> u64 {
    unsafe { core::arch::x86_64::_mm_mfence() };
    let tsc = unsafe { core::arch::x86_64::_rdtsc() };
    unsafe { core::arch::x86_64::_mm_mfence() };
    tsc
}
pub unsafe fn only_flush(p: *const u8) -> u64 {
    let t = unsafe { rdtsc_fence() };
    unsafe { core::arch::x86_64::_mm_clflush(p) };
    (unsafe { rdtsc_fence() } - t)
}
pub unsafe fn only_reload(p: *const u8) -> u64 {
    let t = unsafe { rdtsc_fence() };
    unsafe { core::ptr::read_volatile(p) };
    (unsafe { rdtsc_fence() } - t)
}
```

Listing 6.1.: Measurement primitives.

We thus implement three strategies:

1. `SingleFlush`: In this strategy, a single address is probed with `clflush` after a run of the pattern, and 64 times more iterations are required to cover the full page.

2. `SingleReload`: This is the same strategy using reload operations as the measurement primitive. It is the only strategy usable with loads, as more measurement loads would interfere with the prefetcher operation. This is the baseline approach used in the literature.

3. `FullFlush`: This is the most efficient technique that flushes the whole page(s) after the pattern has been run.

The `SingleFlush` method enables comparison with `SingleReload` and the `FullFlush` methods.

We warm up over 100 iterations and collect data over 1024 iterations. Before each pattern run, a fresh page is selected from a shuffled queue to reset the prefetcher state. Then the page is fully flushed. Given Rohan et al. [94] findings and the Intel manuals [46, 47], cycling through a larger number of pages should evict any lingering entry and reset the prefetcher state for the page. The Prober is configured to cycle over 63 pages, which should reset the prefetcher state between page re-use.

Modern CPU's out-of-order execution means extra serialization is needed to ensure the correct measurement of execution times. Listing 6.1 presents the measurement code. `fn only_flush` is used for `SingleFlush` and `FullFlush`, while `fn only_reload` is used for `SingleReload`.

**Takeaway:** We implement three strategies to measure the final state of the cache resulting from an access pattern and prefetcher activity. `FullFlush` is the optimized method using Flush+Flush, `SingleReload` is the much slower method using Flush+Reload, and `SingleFlush` is comparable with either method. The `Full` method checks the status of every line in the page after each pattern run, while the `Single` methods measure a single line and repeat experiments 64 times. In all three methods, we serialize the measurements using `mfence`[a].

[a] It was later discovered that these fences prevented the L1 prefetchers from activating.

### 6.2.2. Access patterns

We study L2 caches, which deal with memory accesses at cache line granularity. We thus identify accesses as cache line offsets within a page, comprised between 0 and 63. An access pattern combines such an offset with a specific function used to make the access. This feature is meant to study prefetchers that take into account the Instruction Pointers (IP) of memory accesses.

Given the size of the pattern space, the following experiments study a restricted subset of all possible patterns.

**Experiment** 0**:** No access, plots a 1D graphic, expecting full misses, which was verified.

**Experiment** 1**:** One access, $(i)$, 2D graphic $(i, p)$ of the hit rate per line in the page, given the one line accessed.

**Experiment** 2**:** Exhaustive exploration patterns of 2 accesses $(i, j)$. The result is a cube of data indexed by $(i, j, p)$.

**Experiments** $A_k$**,** $k \in \{1, 2, 3, 4, 8\}$**:** Sequence of $(i, i + k, j)$. The result of each experiment is a cube $(i, j, p)$.

**Experiments** $B_k$**,** $k \in \{1, 2, 3, 4, 8\}$**:** Sequence of $(i, i - k, j)$. The result of each experiment is a cube $(i, j, p)$.

**Experiments** $C_k$**,** $k \in \{1, 2, 3, 4, 8\}$**:** Sequence of $(i, j, j + k)$. The result of each experiment is a cube $(i, j, p)$.

**Experiments** $D_k$**,** $k \in \{1, 2, 3, 4, 8\}$**:** Sequence of $(i, j, j - k)$. The result of each experiment is a cube $(i, j, p)$.

**Experiments** $E_k$**,** $k \in [\![2, 4]\!]$**:** $(i + n \times j)_{n \in [0,k]}$ strides. The result of each experiment is a cube $(i, j, p)$.

**Experiments** $F_k$**,** $k \in \{-4, -3, -2, -1, 1, 2, 3, 4\}$**:** Sequence of $(i, i + k, j, i + 2k)$. The result of each experiment is a cube $(i, j, p)$.

Experiments 0 and 1 aside, the experiments result in cubes $(i, j, p)$, giving for each pattern $f(i, j)$ the hit rate of each line (probe $p$) in the page. As cubes cannot be easily visualized in 2D, we derived two kinds of figures from them:

**Total prefetch in page** (e.g., Fig. 6.1a, page 104): The cube is flattened into a 2D heatmap that for $(i, j)$ gives the mean number of prefetches in the page (sum along the third axis).

**Slices** (e.g., Fig. 6.1b, page 104): Heatmap indexed by $(j, p)$ for a set $i$.

We designed experiments $A$ to $F$ to further study the Stream prefetcher, using the insights from experiments 1 and 2. Additional patterns could be designed to study other prefetchers or behaviors. Figures 6.1 and 6.3 to 6.8, pages 104 and 106 to 111, present a selection of our results. The full data set will be available online.

To understand figures with any prefetcher enabled, one must first identify the motif of hits from the pattern itself; additional hits are prefetches. Flattened cubes only give the total number of prefetches in the page but help choose slices to study.

In figures where the $y$ axis is $p$ (probe), accesses that depend only on fixed parameters (e.g., $i = 15$) will cause horizontal lines of hits. Meanwhile, accesses in the access pattern that depend on the $x$-axis will cause diagonal lines (e.g., access $j + 1$ when the $x$-axis is $j$). To make this clearer, we go through a pair of slice figures, with no prefetchers enabled: $E_4$ and $D_4$:

- $E_4$ executes the pattern $(i, i + j, i + 2j, i + 3j, i + 4j)$. Excluding prefetches, each run results in 5 hits in the page (or less for values of $j$ like 0 or 32 where some of the accesses coincide). Thus, each column of the figure contains up to 5 points. On the slice with $i = 15$, Fig. 6.1b, page 104, the horizontal line corresponds to the first access, the diagonal at an offset corresponds to $i + j$, and further even more slanted lines of dots corresponds to $i + kj$ with higher $k$.
- In $D_4$, slice $i = 14$, Fig. 6.1c, page 104, we observe a horizontal line for $p = 14$, and two diagonal lines, one for $p = j$ and one for $p = j - 4$. They correspond respectively to the first, second, and third access of the pattern $(i, j, j - 4)$.

With these motifs in mind, we can easily see the prefetch response to various patterns, answering **Q6.1**: Prefetched lines are any high hit rate lines not in the pattern motif.

> **Takeaway:** We collected the results for families of patterns called experiments, identified as 0, 1, 2, $A_k$ ($A_1$ to $A_8$), $B_k$, $C_k$, $D_k$, $E_k$ and $F_k$, and described above. All experimental results aside from 0 and 1 take the shape of a cube. Thus, most figures in this chapter are *slices* of those cubes.

## 6.3. Experimental setup

### 6.3.1. Hardware and software configuration

We use the two machines described in Section 4.5, identified as Wkl and Cfl. Both CPUs are minor refreshes of the Kaby Lake architecture. However, our results suggest their prefetchers differ slightly.

### 6.3.2. Control group

To ensure the prefetcher causes our observations, we run identical experiments with all prefetchers disabled using MSR 420. Figures 6.1a to 6.1c, page 104, show that these results exhibit no

cache hit aside the pattern itself and random noise. Our later observations are thus solely due to the prefetcher we enable.

> **Takeaway:** Our observations show that with no prefetcher enabled, only the memory accesses that constitute the pattern exhibit hits. Additional hits observed can thus be attributed to the prefetchers' activity with high confidence.

## 6.4. The L2 Stream prefetcher

### 6.4.1. Proposed data structure for the prefetcher

We assume that this prefetcher ignores offset bits used to select bytes inside lines and uses physical addresses by virtue of operating on the L2 cache. We thus identify cache lines in a page with numbers in $[\![0, 63]\!]$.

Given our results, we propose the following structure, illustrated in Fig. 6.2:

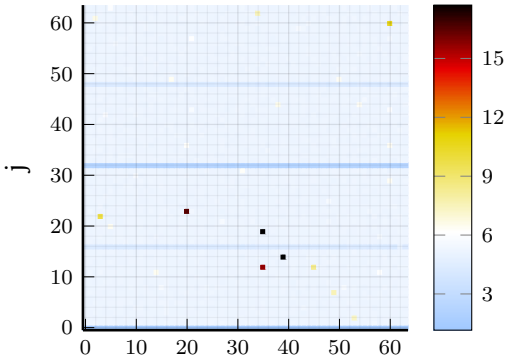
- A table of *stream entries*, as suggested by [94], with entries tagged by a page number. Each entry contains the *last fetched line* in the page, a direction state, and a prefetcher confidence state. We have not determined the structure of these states. The *last fetched line* ($L$) is the last prefetch candidate or the last request when lacking such a candidate.
- A *prefetch candidate* (❶) logic block taking as an input the miss from the L1 cache and the stream entry for the same page. It suggests prefetch candidates and updates the stream entry, using the stream direction state and *last fetched line*.
- A *prefetch confidence* (❷) logic block that computes and updates the confidence prefetcher for this stream, based on requests made and whether they fit the stream.
- A *prefetch arbitration* (❸) logic using the confidence and candidates from all prefetchers to decide whether to issue them.

While incomplete, this structure seems consistent with our observations and explains behaviors observed by [94].

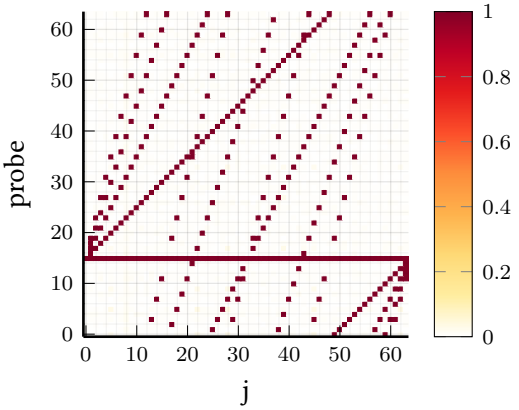
> **Takeaway:** We surmise the Stream prefetcher implementation has the following key features:
>
> - A table, tagged by PPN, stores entries including a stream direction, the last line in the stream, and a confidence metric.
> - A separate logic decides whether to issue the prefetch from the logic determining candidates.

### 6.4.2. Experimental results

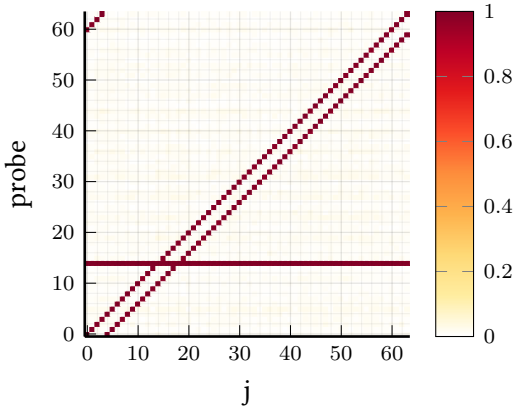
While the Whiskey and Coffee Lake CPUs differ in detail, the same general principles apply. However, the prefetch arbitration in Coffee Lake seems to prefetches more aggressively than the Whiskey Lake one. This prefetcher fetches from main memory into L2 and, by inclusivity,



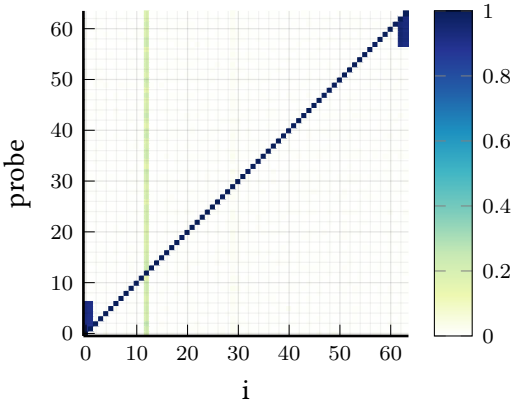

(a) WKL, exp. $E_4$, $(i, i+j, \ldots, i+4j)$ with no prefetcher: average number of hit in the page for $i$ and $j$.



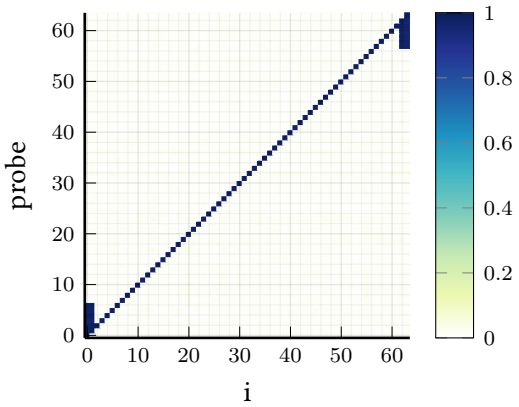

(b) WKL, exp. $E_4$, $(i, i+j, \ldots, i+4j)$ with no prefetcher: hit rate in each line in the page for $j$, with $i = 15$.



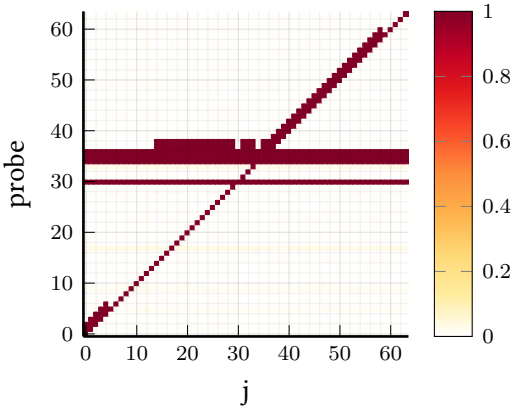

(c) CFL, exp. $D_4$, $(i, j, j-4)$ with no prefetcher: hit rate in each line in the page for $j$, with $i = 14$.




(d) WKL, exp. 1, $(i)$ with Stream prefetcher: hit rate in each line in the page for $i$.




(e) CFL, exp. 1, $(i)$ with Stream prefetcher: hit rate in each line in the page for $i$.




(f) WKL, exp. $A_4$, $(i, i+4, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 30$.

Figure 6.1.: Experimental results, see Section 6.2.2 for the description of the various experiments. Different color scales are used to identify better the various representations (cube summation (blue to red) vs. cube slice (white to red) vs. full experiment (white to blue)). Unless specified otherwise, the method used is `FullFlush`.

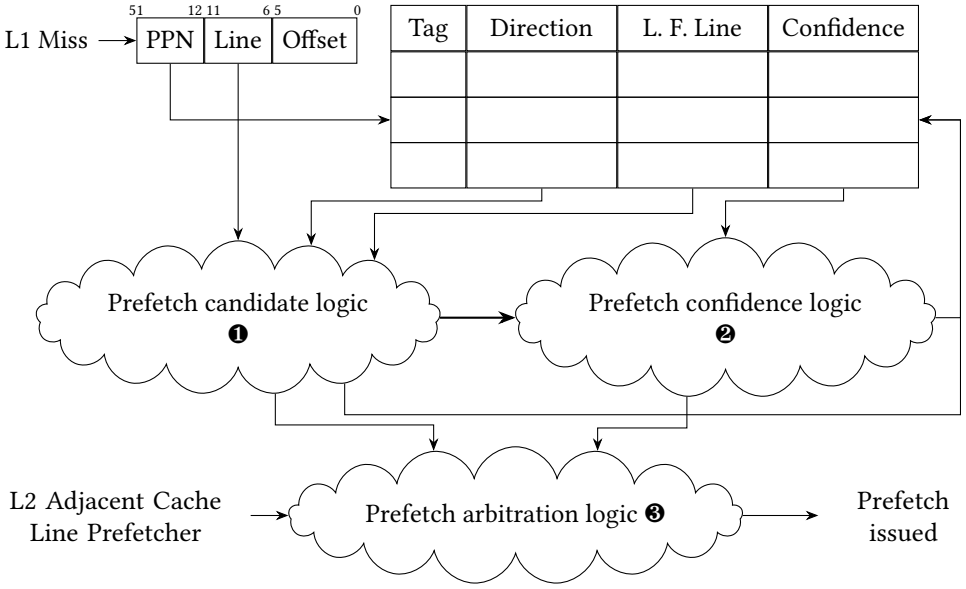



Figure 6.2.: The proposed structure for the Stream prefetcher.

L3, or sometimes only L3. In addition, Whiskey Lake has a smaller L3 cache. Thus, the L3 size cannot be excluded as a reason for such differences.

> **Takeaway:** Whiskey and Coffee Lake prefetchers appear to be close but not identical. The difference in L3 size may explain this.

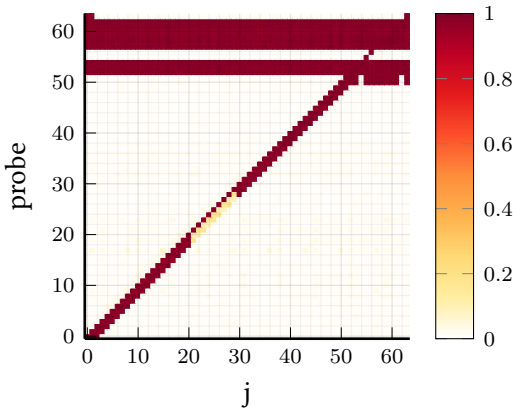
#### First access behavior

Prefetch may occur if the first access in a page is in $\{0, 1, 62, 63\}$, (first and last two lines of a page); see Figs. 6.1d and 6.1e, page 104. Lines upward to line 6 (resp. downward to line 57) are then fetched, and the *last fetched line* is set to 6 (resp. 57). Otherwise, no prefetch occurs on the first access; the *last fetched line* is set to the line accessed. On Whiskey Lake, this also occurs in a minority (10%) of first accesses in $\{0, 1, 62, 63\}$ and behaves like first accesses in lines 2 or 61. On Coffee Lake, this prefetch always occurs; see Fig. 6.1e, page 104.

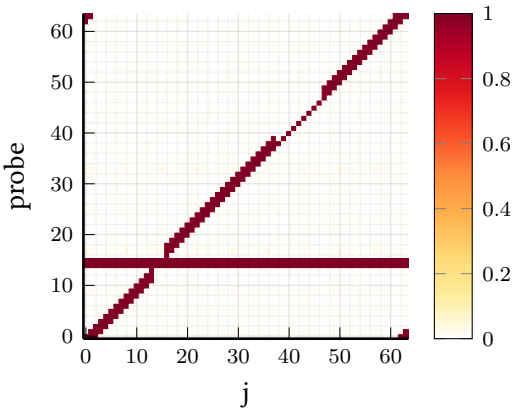
> **Takeaway:** The first two and last two lines of a page are treated differently by the Stream prefetcher when the first access to a given page occurs on these lines. One such access usually starts a Stream without further confirmation and prefetches up to 6 consecutive lines.

#### Subsequent accesses prefetch 0 or 2 lines

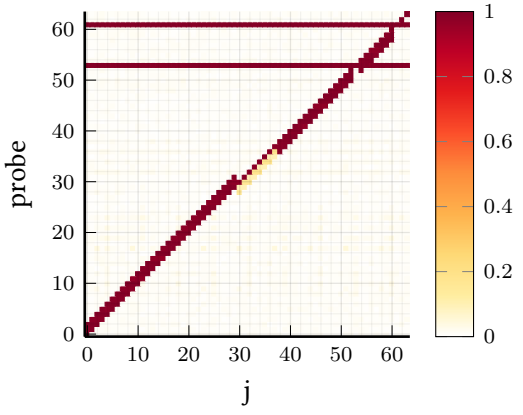
On subsequent accesses, 0 or 2 consecutive lines are prefetched adjacent to either the *last fetched line* or the current access, as can be observed for results in Figs. 6.1 and 6.3 to 6.8, pages 104 and 106 to 111, where only the Stream prefetcher is enabled. Section 3.3 of [94] claims the prefetch distance is 1 to 4, and the degree is 4 to 8. However, our results show several prefetches



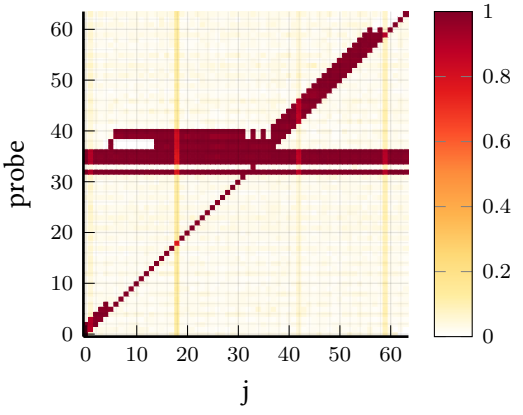

(a) CFL, exp. $B_8$, $(i, i-8, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 62$.



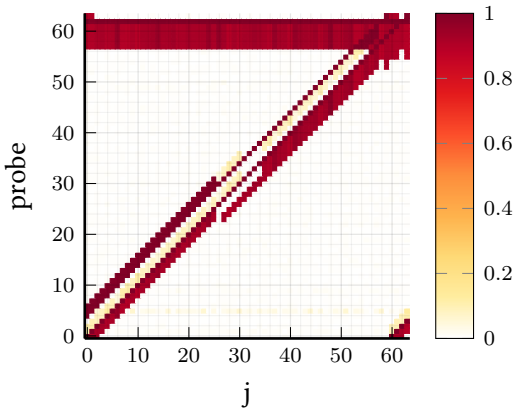

(b) WKL, exp. $B_1$, $(i, i-1, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 15$.



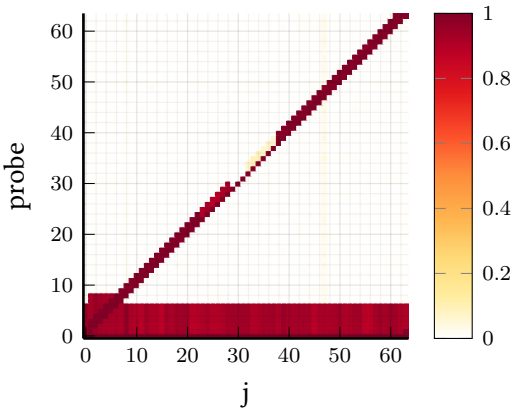

(c) CFL, exp. $B_8$, $(i, i-8, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 61$.




(d) WKL, exp. $F_2$, $(i, i+2, j, i+4)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 32$.

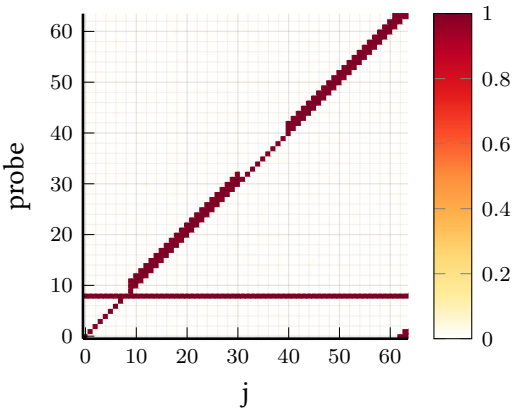



(e) WKL, exp. $C_4$, $(i, j, j+4)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 62$.

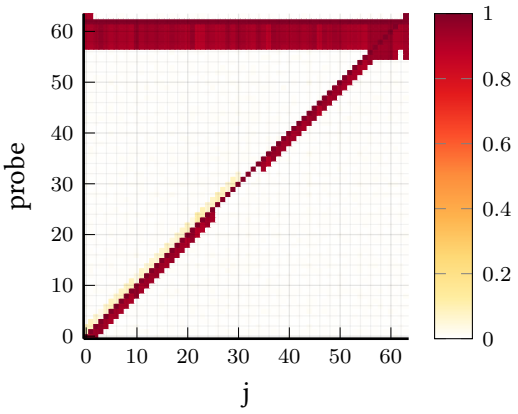



(f) WKL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 0$.

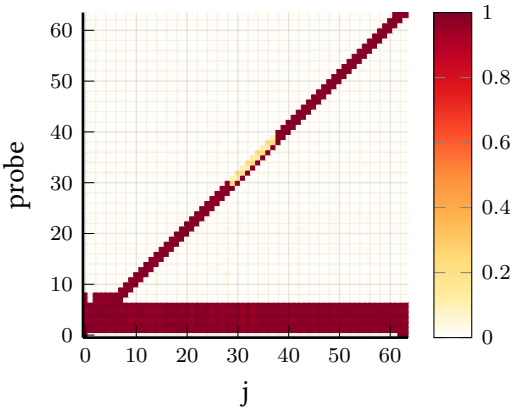
Figure 6.3.: Experimental results continued from Fig. 6.1. See Section 6.2.2 for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.



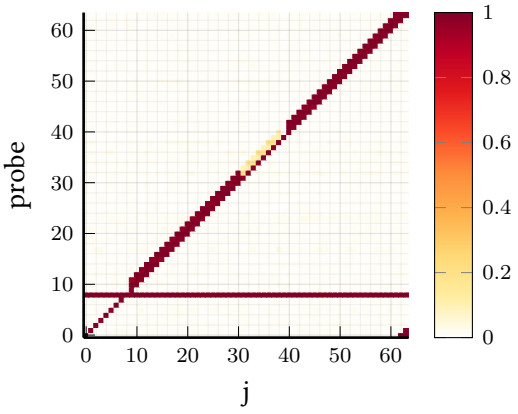

(a) WKL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 8$.



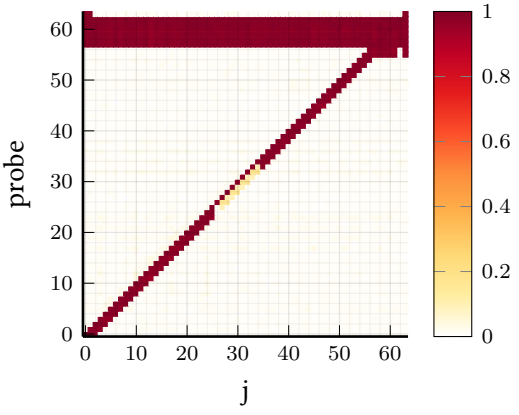

(b) WKL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 62$.



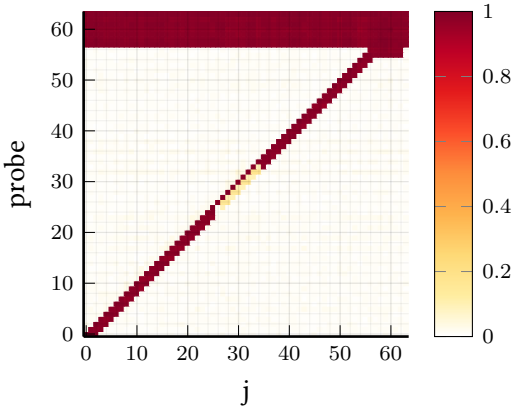

(c) CFL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 1$.




(d) CFL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 8$.

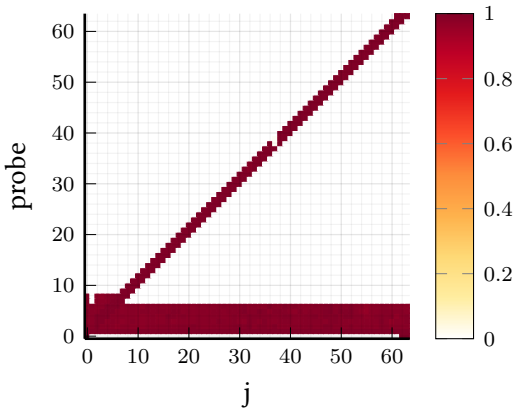



(e) CFL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 62$.

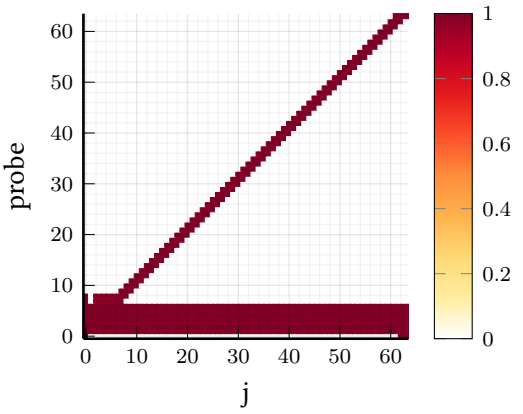



(f) CFL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 63$.

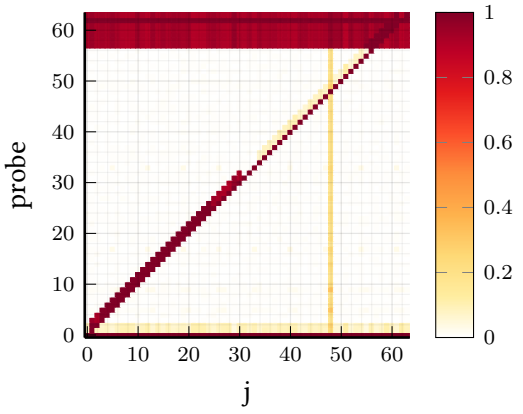
Figure 6.4.: Experimental results continued from Fig. 6.3. See Section 6.2.2 for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.



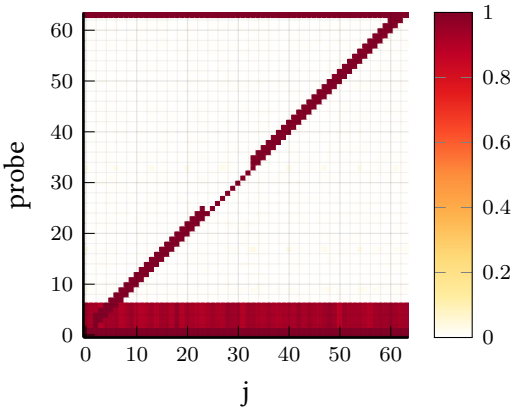

(a) `SingleReload` WKL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 1$.



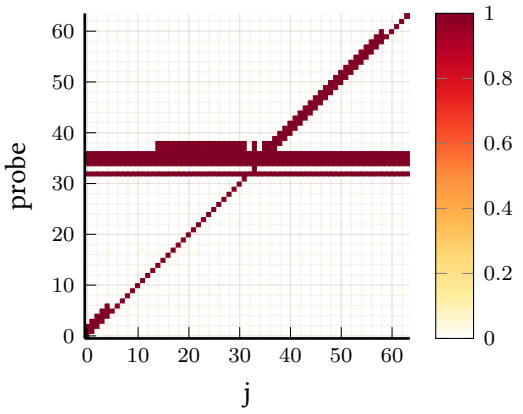

(b) `SingleReload` CFL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 1$.



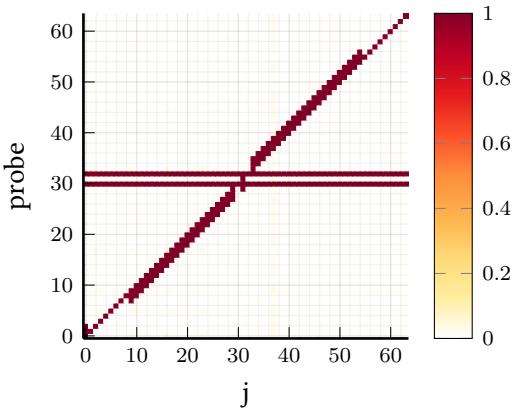

(c) WKL, exp. $A_2$, $(i, i+2, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 62$.




(d) WKL, exp. $B_2$, $(i, i-2, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 1$.

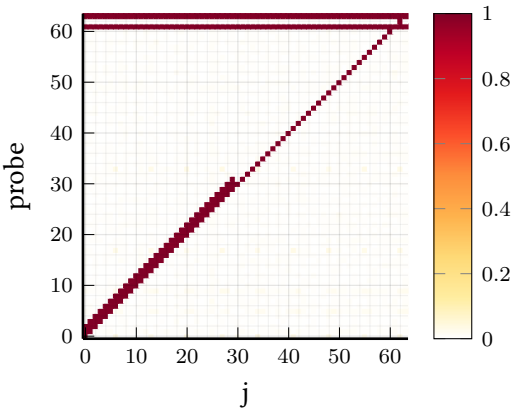



(e) WKL, exp. $A_2$, $(i, i+2, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 32$.

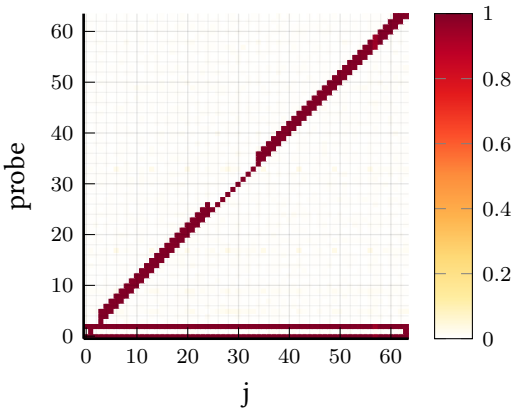



(f) WKL, exp. $B_2$, $(i, i-2, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 32$.

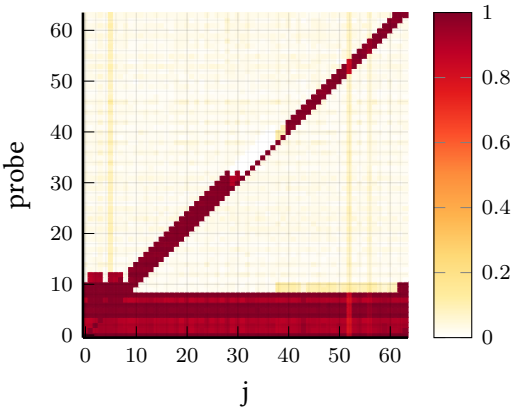
Figure 6.5.: Experimental results continued from Fig. 6.4. See Section 6.2.2 for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.



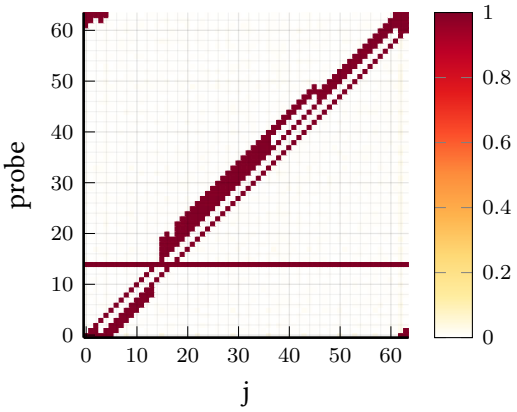

(a) Wkl, exp. $A_2, (i, i+2, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 61$.



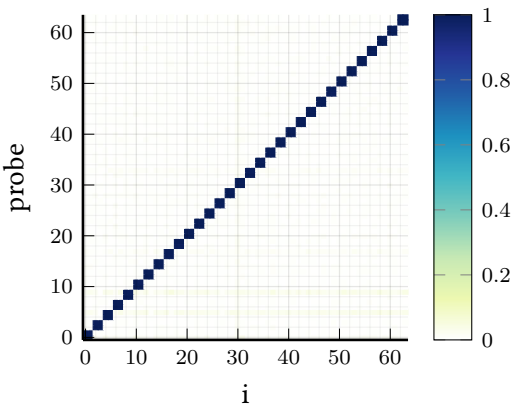

(b) Wkl, exp. $B_2, (i, i-2, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 2$.



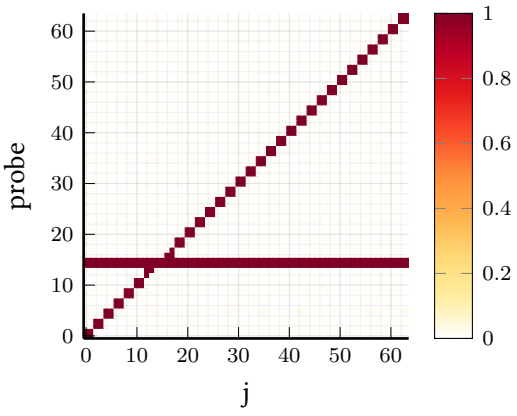

(c) Wkl, exp. $F_2, (i, i+2, j, i+4)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 0$.




(d) Wkl, exp. $D_3, (i, j, j-3)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 14$.

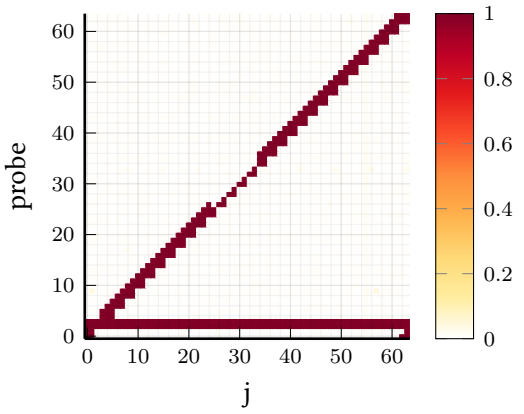



(e) Wkl, exp. $1, (i)$, with adjacent line prefetcher: hit rate in each line in the page for $i$.

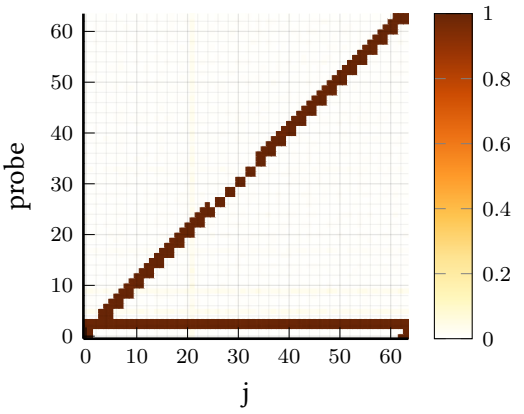



(f) Wkl, exp. $B_1, (i, i-1, j)$, with adjacent line prefetcher: hit rate in each line in the page for $j$, with $i = 15$.

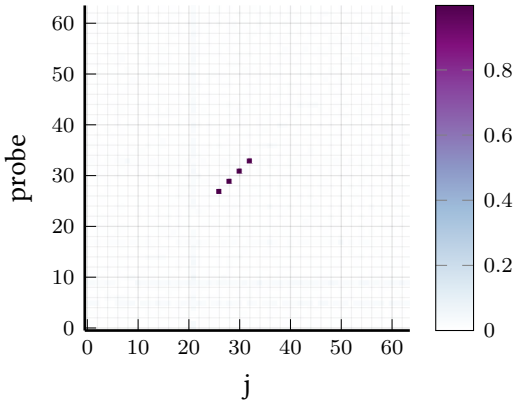
Figure 6.6.: Experimental results continued from Fig. 6.5. See Section 6.2.2 for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.



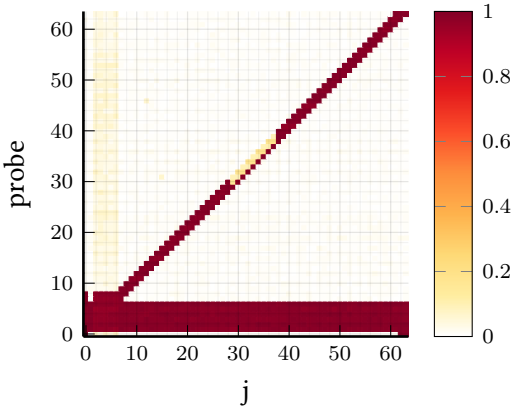

(a) WKL, exp. 2, $(i, j)$, with both prefetcher: hit rate in each line in the page for $j$, with $i = 2$.



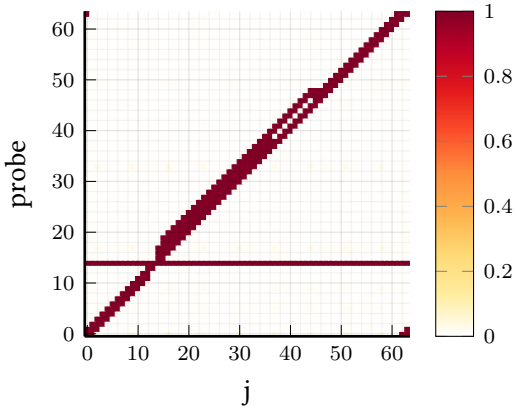

(b) WKL, exp. 2, $(i, j)$, maximum of the results for each prefetcher: max hit rate in each line in the page for $j$, with $i = 2$.



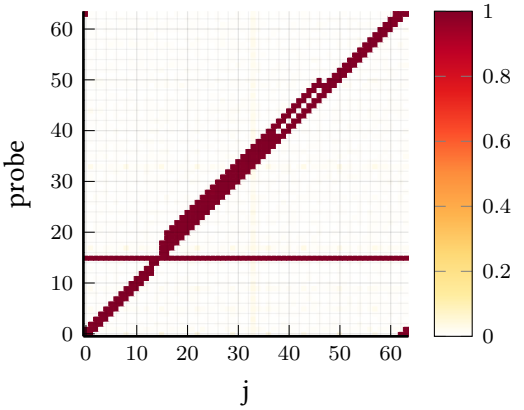

(c) WKL, exp. 2, $(i, j)$, difference between the two previous figures in hit rate in each line in the page for $j$, with $i = 2$.




(d) `SingleFlush` CFL, exp. 2, $(i, j)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 1$.




(e) WKL, exp. $C_1$, $(i, j, j+1)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 14$.




(f) WKL, exp. $C_1$, $(i, j, j+1)$, with Stream prefetcher: hit rate in each line in the page for $j$, with $i = 15$.

Figure 6.7.: Experimental results continued from Fig. 6.6. See Section 6.2.2 for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.

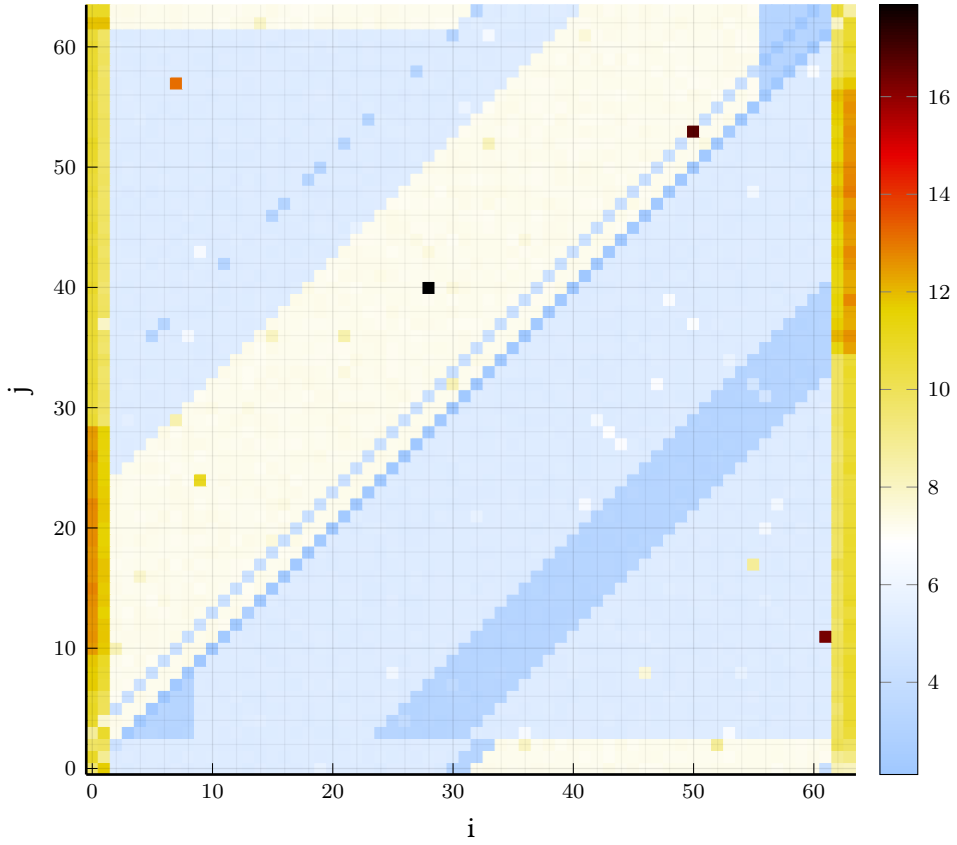



Figure 6.8.: WKL, exp. $D_3$, $(i, j, j-3)$ with Stream prefetcher: average number of hit in the page for $i$ and $j$.

may occur with their access pattern, $(0, 1, 2, 3)$ in our notation. Their upper bound arises from prefetches to lines 1 to 6 on the first access, followed by prefetches to 7 and 8, 9 and 10, and then 11 and 12 on the subsequent 3 accesses. This insight is enabled by measuring the precise effect of patterns on the cache and the impact of their prefixes to determine which access is causing a given prefetch.

> **Takeaway:** Aside from the previous edge case, the Stream prefetcher requests zero or two lines.

#### Location of prefetches

As seen in Figs. 6.1f and 6.3a, pages 104 and 106, prefetches occur in either a positive or negative direction. A positive prefetch fetches $max(current, L) + 1$, $max(current, L) + 2$ where $current$ is the line being accessed and $L$ the *last fetched line*. A negative prefetch instead fetches $min(current, L) - 1$ and $min(current, L) - 2$. This logic helps with out-of-order accesses, which the cache may observe due to out-of-order execution.

> **Takeaway:** The aforementioned two prefetches occur from the *last fetched line* or the current address, whichever is the further along the stream direction.

#### Stream window and stream reset

This arithmetic is done modulus 64, and thus prefetch wraps around pages when nearing a page boundary, as seen in Fig. 6.3b, page 106, $(B_1, i = 15)$, with $j \in \{0, 1, 62, 63\}$. This is a safe guess (as long as no finer access control granularity is introduced), albeit likely useless.

However, accesses more than 31 cache lines away are deemed by the prefetcher to no longer be part of the stream and reset its stream entry. In that case, the direction is updated towards the positive direction, and two prefetches are issued from the current address with the updated direction. This is visible in Figs. 6.3a, 6.3c and 6.3d, page 106. For a first access in line 62 or 63, this is a barely negative direction (Fig. 6.3e shows an extra access is needed to get a positive prefetch), but other experiments result in positive prefetches even if the previous one was negative and we loaded a line below the *last fetched line*. This suggests that accesses are only deemed to belong in a stream if they are in a *window* of 62 lines around the *last fetched line*. [112] includes such a window.

> **Takeaway:** The prefetch candidate arithmetic is computed modulus 64; hence prefetch wraps around page limits. In addition, only accesses within a *window* of $\pm 31$ line around the *last fetched line* are part of a stream; accesses further seem to cause a reset.

#### Prefetch gap

With a `clflush`-based method, as shown in Figs. 6.3f and 6.4a to 6.4f (pages 106 and 107), we observe *a prefetch gap* in the area with $j \in [\![L + 23, L + 31]\!]$ with $L$ the *last fetched line*. In this area, no prefetches are issued on Whiskey Lake, whereas they do occur in a minority of cases on Coffee Lake. This is one of the main differences between the Stream prefetchers in Coffee and Whiskey Lake.

On Whiskey Lake, Fig. 6.3f ($i = 0$) shows a superposition of two modes, a minor mode with a 23–31 *gap*, consistent with no first access prefetch, and a majority mode with a 29–37 *prefetch gap*, consistent with a first access prefetch until 6. Thus, the gap starts on the 23rd line from the *last fetched line*, forward and backward as seen in Fig. 6.1f (page 104), and ends on a stream resetting access outside the *window* defined above.

However, the same lack of prefetch disappears with `SingleReload`, except for $L+31$ discussed later, as seen in Figs. 6.5a and 6.5b, page 108. This shows that the prefetch behavior adapts to the prefetcher's success rate.

> **Takeaway:** For accesses between 23 and 31 lines away from the *last fetched line*, the Stream prefetcher is much less eager to issue prefetches. The upper limit corresponds to the edge of the stream window defined above. We call this area a prefetch *gap*.

### Prefetch direction

We have not fully elucidated the prefetch direction logic, but we can make the following observations. The behavior on a first access in lines 0, 1, 62, or 63 are special cases and set the stream direction directly to positive (for the former two) or negative (for the latter two), as seen in Figs. 6.5c and 6.5d, page 108. Otherwise, there is a bias toward positive prefetch: an extra access is required to get a negative prefetch. Accesses outside the stream window will update the direction towards positive and immediately issue prefetches, as explained above and seen in Figs. 6.5e and 6.5f.

Additionally, the prefetcher is reluctant to start a positive stream for access in lines 56 to 61, and similarly, a negative one from lines 8 to 2, as shown by Figs. 6.6a and 6.6b, page 109.

> **Takeaway:** The prefetcher direction state machine is not a single-bit system and has special cases for page edges and against initiating streams too close to the end of a page in either direction.

### Prefetch may occur on L2 hits

Looking at Fig. 6.3d, page 106 ($F2$, $i = 32$), an access to a line already prefetched in L2 triggers further prefetches. In this instance, the last access of the pattern is $i+4 = 36$, and for $j \in [5, 56]$, we observe a prefetch on this access. The figure is best interpreted by comparing it with Fig. 6.5e (page 108) whose pattern is the underlined prefix of $F_2$: $(\underline{i, i+2, j}, i+4)$.

It is worth noting that such an access must still be part of the aforementioned prefetch window, as shown by Fig. 6.6c, page 109, where $j \in [\![40, 61]\!]$ do not exhibit prefetches on the fourth access.

> **Takeaway:** We can confirm prefetch may be issued on L2 hits.

### Suppressed prefetches may still update the prefetcher state

In Fig. 6.6d (page 109), when $j \in [\![37, 45]\!]$, we observe prefetches occurring for $j + 3$ and $j + 4$ without prefetch issued for $j + 1$ and $j + 2$. This appears surprising but makes sense in our model if the arbitration logic suppresses the prefetch but the candidate logic (and confidence

logic) still updates the stream table, with a *last fetched line* of $j + 2$. The second access 3 lines below is within the stream window, increases the confidence, and thus triggers prefetches from the *last fetched line* ($j + 2$), even though the $j + 1$ and $j + 2$ prefetches were suppressed for insufficient confidence. This is why we proposed a split of the prefetch arbitration logic (❸) in our model.

> **Takeaway:** Behavior in the prefetch gap suggests the prefetcher internal state can be updated by accesses that did not cause observable changes in the cache in a way that is consistent with a suppressed prefetch.

#### Summary

While some uncertainties remain, we get a far more precise picture of this prefetcher's behavior, and our framework can provide further insights using new patterns.

The Stream prefetcher treats in a particular way streams that first access a page in its first or last two lines. Otherwise, if confident enough, it prefetches a pair of consecutive lines starting on the *last fetched line* or the current line, whichever is furthest along the direction of the stream. Accesses 32 or more lines away from the *last fetched line* are treated differently. Furthermore, prefetches issued will safely wrap around page limits, which may issue pointless prefetches but causes no potentially dangerous prefetches across page limits.

The prefetcher outputs a confidence metric that contributes to the arbitration of whether to prefetch, but suppressed prefetches may cause updates to the prefetcher state. Lastly, the prefetcher is reluctant to start streams with little room until the page limit. These results significantly add to the previous work about **Q6.3**.

> **Takeaway:** In this section, we have documented several behaviors exhibited by the L2 Stream prefetcher in Intel Coffee and Whiskey Lake CPUs, consistent with the structure we proposed.

## 6.5. The L2 Adjacent Cache line prefetcher and prefetcher interaction

The adjacent cache line prefetcher is the other L2 prefetcher disclosed by Intel, thought to treat cache lines as 128-byte pairs, fetching the sibling line upon access to the other. On a first access in a page, its behavior (Fig. 6.6e, page 109) matches this description. However small differences appear with more accesses, e.g., in Fig. 6.6f, page 109, the expected prefetch is missing for $j = 13$ or 16. Suggestively, compared with the same figure (6.3b, page 106) with the Stream prefetcher enabled, the missing prefetch is a line that the Stream prefetcher would have prefetched.

To check for prefetcher interferences, we first run experiments with both prefetchers enabled. In Fig. 6.7a, page 110, it appears the two prefetchers do not use one another requests as input, which justifies comparing it with the superposition of individual experiments, shown in Fig. 6.7b. Differences appear when comparing the superposition with the experiment with both prefetchers enabled, as demonstrated in Fig. 6.7c. In areas where arbitration suppresses

the Stream prefetches, the adjacent cache line is not prefetched if it coincides with a suppressed prefetch.

The details require further study, but this answers **Q6.4** and shows that studying prefetchers in isolation is insufficient.

> **Takeaway:** The L2 Adjacent cache line prefetcher exhibits unexpected behaviors when consecutive addresses are accessed and interactions with the L2 Stream prefetcher.

## 6.6. Discussion

### 6.6.1. Advantages of using `clflush` compared with a Load based technique

`clflush` evicts lines without using them, whereas a load vindicates the prefetcher in its guess. The latter may increase the prefetcher's confidence and/or the arbitration logic trust in the prefetcher, akin to tournament branch predictors [41], thereby interfering with measurements.

Our experiments show clear differences between measurements with `SingleReload` (e.g., Fig. 6.5b, page 108) and those with `clflush` (Figs. 6.4c and 6.7d, pages 107 and 110). Using `clflush` has also allowed us to detect prefetcher state updates on no observed prefetch. This is why we proposed the prefetcher arbitration logic (❸) similar to tournament branch predictors. This also showed that the distance to the *last fetched line* impacts this confidence.

In addition, we observe that `FullFlush` (Fig. 6.4c) and `SingleFlush` (Fig. 6.7d) give the same results, this shows it is safe to measure the whole page using `FullFlush`, unlike `SingleReload`, limited to one line per execution of the pattern, which multiplies the number of pattern runs by the number of lines in the page ($\times 64$ on Intel CPUs).

Consequently, we can answer **Q6.2** in that measuring with loads creates interferences compared to measuring with `clflush`. Therefore, Flush+Flush is both a faster and a more subtle tool for prefetcher reverse engineering than Flush+Reload and provides more insights.

> **Takeaway:** A Flush+Flush-based study of prefetcher interferes less with the prefetcher and shows that even loading a single address as part of the measurement can interfere with the prefetcher behavior. Combining the `FullFlush` and `SingleReload` approaches shows that the Stream prefetcher has some adaptability.

### 6.6.2. Areas of uncertainty

While we now understand the Intel L2 prefetchers better, several areas of uncertainty remain.

First, the behavior when an access occurs at $L + 31$ appears unpredictable. Figure 6.8, page 111, shows that the number of prefetches observed along the diagonal $j = i + 31$ is inconsistent. Furthermore, even with an equal number of prefetches, they sometimes occur in different places, e.g., in Figs. 6.7e and 6.7f (resp. $i = 14$ and 15), page 110, the behavior for $L + 31$, (45 and 46) is different, even if they have the same number of prefetches. However, the pattern behaves consistently over repeated runs.

Secondly, on Whiskey Lake, we have not found the source of the two modes for first accesses in $\{0, 1, 62, 63\}$. We show that 90% of the time, we get prefetches of lines up to 6 or down to 57.

Meanwhile, we get a behavior consistent with lines 2 and 61 the remainder 10% of the time. We do not know the source of this split.

In addition, the stream direction state machine has not been determined yet, even if we now know it is biased in favor of positive streams and that to start negative streams, an extra access is necessary or a first access in the page to line 62 or 63.

Finally, the confidence logic, which arbitrates whether to actually prefetch the line and arbitrates between the various prefetchers, requires simultaneous investigation of both prefetchers.

> **Takeaway:** Our study has not been able to fully reverse engineer the algorithm of the Stream prefetcher, and some areas of unexplained behavior exist.

### 6.6.3. Limitations

CacheObserver is not able to observe L1 prefetches[3]. It might be that the L1 prefetchers cannot fetch from main memory but only fetch from L2 or L3. A solution may be to calibrate reloads to determine in which level a line is cached.

Our framework has information about which access causes which prefetch but has no temporal information on when prefetches are issued and handled. This, for instance, precludes understanding how exactly 2 or more requests are emitted per L2 access. For example, those could be emitted over several consecutive cycles, but this is not necessarily the design used.

> **Takeaway:** We are currently unable to study L1 prefetchers and do not have any temporal information as to when prefetches are issued and completed.

## 6.7. Conclusion and future work

We developed *CacheObserver*, a framework that leverages a carefully calibrated Flush+Flush side channel, based on `clflush`, to get a detailed view of the prefetcher activity in reaction to access patterns. This allowed us to model the L2 Stream prefetcher and uncover a variety of its behaviors. Our experiments showed that prefetchers behave differently under `clflush` measurements compared to reload measurements used in the state of the art, which interfere with prefetcher activity. In addition, our technique requires fewer measurements, as we can measure a whole page in one go with `clflush`, compared to repeated experiments for each line of the page. Finally, we uncovered interactions between the two L2 prefetchers, especially when a line is a candidate for both.

We were thus able to answer **Q6.1** to **Q6.4** in the following way:

- **Q6.1** *How can we build a detailed view of prefetch activity?* Prefetched lines are any high hit rate lines not in the pattern motif (see Section 6.2.2).
- **Q6.2** *What is the impact of using load instructions for measurements on prefetch activity compared with* `clflush`*?* Measuring with loads creates interferences compared to measuring with `clflush` (see Section 6.6.1).

[3]We have more recently found indication in the documentation that our use of fences could explain this.

- **Q6.3** *How does the Intel L2 Stream prefetcher behave, especially on the first few accesses in a page?* Section 6.4.2 (pages 103 to 114) presents our findings regarding the behaviour of the Intel L2 Stream prefetcher.
- **Q6.4** *Do the various prefetchers in Intel CPUs interact?* Section 6.5 shows that there are interactions between the two L2 prefetchers.

Within the wider scope of this thesis, we answer **Q1** and **Q2**, as defined in Section 2 of the Introduction, in the following way:

- **Q1**: *How can we determine what is the exact effect of the prefetcher on the cache in reaction to a given sequence of memory accesses?* The *CacheObserver* can monitor a range of addresses and identify the status of every line within the range after executing a sequence of memory accesses. Prefetches can be observed as cache hits on lines that were not accessed. It is usually necessary to also acquire the results of prefixes of the sequence to distinguish what memory access caused each prefetch.
- **Q2**: *What cache primitive is more appropriate to reverse engineering prefetchers?* Given our much more performant results and the different behavior observed, we can conclude that Flush+Flush is a better primitive. Still, combining Flush+Flush with extra targeted Flush+Reload measurements provides additional insights.

More generally, this framework can also help investigate and optimize the performance of programs by getting detailed information on how the memory system — caches and prefetchers — reacts to those programs' memory requests. For instance, this can validate hypotheses as to whether prefetchers are effectively prefetching the data required by a program and what memory accesses are not getting prefetched.

### Further work

A first research direction could be load calibration to distinguish hits from L1, L2, and L3, in addition to removing the fences that inhibit the L1 prefetcher. This would enable the study of the L1 prefetchers. A second research direction could be to study the state machine for stream directions to determine its precise structure, as well as the structures used for prefetcher confidence (❷) and arbitration (❸), by building higher levels of abstraction, as done by Vila et al. [123].

# 7. The Dendrobates framework

In this chapter, we go over the architecture of the *dendrobates* framework, which results from the various works presented in Chapters 4 to 6. We hope that this documentation will make the re-use of this code easier. As mentioned in Chapter 4, this framework started as a bare-metal measurement platform and was later used as the basis for the hosted-mode codebases for the two papers.

We will first cover the general architecture (Section 7.1), then focus in turn on the topology-aware calibration code (Section 7.2), which underpins Chapter 5, and the bare-metal support (Section 7.3). We will then discuss the various abstractions (Section 7.4) we wrote followed by the concrete implementations of the Flush+Reload and Flush+Flush implementing those (Section 7.5). This will lead to the discussion of the channel benchmarks (Section 7.6) and of the CacheObserver framework (Section 7.7), both examples that use the abstractions and their concrete implementations. We will discuss various utilities (Section 7.8) used in parts of the framework before the final discussion of the advantages and limitations of the framework (Section 7.9).

## 7.1. Framework Design

We will first recall the general goals of the framework, combining those listed in Chapter 4 with the additional one resulting from Chapters 5 and 6. We will then explain our decision to write this framework in Rust and, finally, Section 7.1.3 will overview the general architecture of the framework.

### 7.1.1. Framework goals

As stated in Chapter 4, the initial goal was to *precisely measure the various sources of variation of* `clflush` *timing and determine the reliability of the Flush+Flush primitive.* As part of this objective, we aimed to have the ability to control entirely the system, including the virtual to physical address translation and controlling interrupts and preemption. Zero-cost abstractions were thus needed from the start.

After this initial study, we successively added the following goals, with only the need to run in a hosted environment *i.e.*, on top of a Linux operating system:

First, we extended the framework to study precisely the impact of the topology on the execution times of instructions, applied it to `clflush`, and exploited the insights to improve cache channel primitives.

Then, as part of the work in Chapter 5, we added the benchmarking of the improved Flush+Flush compared with other primitives. This included both designing abstraction layers to sim-

plify the use and implementation of such primitives and the implementation of two benchmarks using these interfaces.

Lastly, as part of the work in Chapter 6, we extended the framework to use the same primitives to monitor the changes in the cache in reaction to sequences of memory accesses. This would allow the detection of prefetcher activity and help uncover their undocumented behavior.

When considering these various objectives, the *dendrobates* framework's goals are twofold. On the one hand, the framework should support the study of the microarchitecture on bare-metal and hosted environments. On the other hand, it should provide abstractions to build and use cache channels, both covert and side channels, and apply these to better understand the behavior of the memory system.

> **Takeaway:** The framework's goals have evolved with the research. The framework's final architecture answers the following goals: First, it is a bare-metal and hosted measure platform that provides abstractions to build and use cache channels. It also includes applications of these to benchmark side and covert channels and, finally, study the behavior of the memory hierarchy.

### 7.1.2. Why Rust ?

One of the primary reasons we selected Rust is that we were initially building a bare-metal environment for 64-bit x86. Rust happened to have a modern ecosystem that provided all the necessary code to boot a system to 64-bit (aka long) mode, whereas the ecosystem in C/C++ usually required taking over in 32-bit mode, setting up paging, and doing the mode-switch manually. Rust thus enabled us to get faster to where we needed to go.

In addition, Rust provides modern programming tools, with the `core` and `alloc` crates being usable on embedded environments (aka `no_std`) and providing many utilities such as containers. The standard `HashMap` are not included in `alloc` but they are instantiated from the `hashbrown` crate, which can support `no_std`. The standard `HashMap` only customizes the hash function to use a system-provided source of randomness. Rust also provides modern tools such as generics, closures or first class arrays, and an expressive type system. However, Rust's most specific feature, its memory ownership model, is less useful to our research than in production code, where it eliminates entire classes of bugs.

Despite those high-level features, Rust abstractions are zero-cost and do not add overhead when they are not used. More importantly, Rust includes an escape hatch to its security guarantees, which allows access to pointers and the low-level features we need. This includes the ability to write inline assembly. This escape hatch use can be limited to small portions of code, and the remainder of the code still benefits from the compiler's guarantees.

Given this, we chose to experiment with using Rust for microarchitecture security research. We will discuss the lessons learned from this experiment in Section 7.9.

> **Takeaway:** We picked Rust as a language because it made it easier to run embedded programs in 64-bit x86 and had high-level features that allowed us to build abstractions while retaining the ability to exert the low-level control we needed.

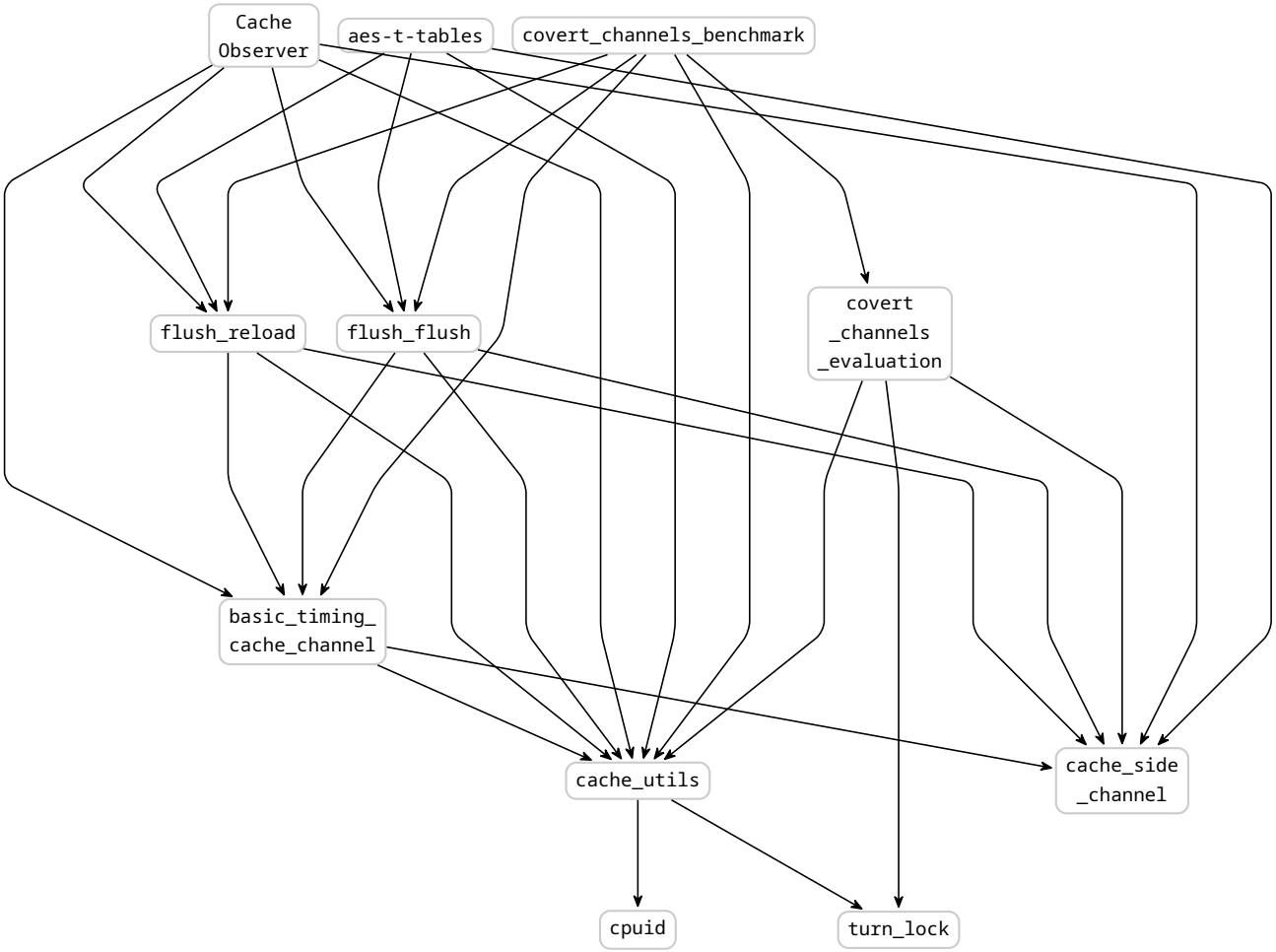



Figure 7.1.: Dependencies between the various crates in the *dendrobates* framework

### 7.1.3. Overall architecture

The framework is divided into a series of modules, called *crates* in Rust terminology. The top level of the workspace corresponds to the bare-metal framework main crate. The `src/` folder thus contains the source of the x86_64 kernel, and `tests` include a few tests for the bare-metal boot. `scripts/` contains scripts used to run the kernel on a simulator for debugging purposes. A `results` folder contains experimental results. The other subfolders each contain one of the dependent crates. Figure 7.1 presents the dependencies between them (excluding the bare-metal support).

They can be grouped into several categories:

**Bare-metal support:** The `polling_serial` and `vga_buffer` crates correspond to the drivers used by our embedded kernel to communicate with the outside world. They are described in Section 7.3.

**Abstractions:** To simplify the implementation of channels and attacks and easily use different attack primitives, we built a series of abstractions. These are the `cache_side_channel` and `basic_timing_cache_channel` crates, along with the crates instantiating Flush+Flush and Flush+Reload, `flush_flush` and `flush_reload`. They are described in Sections 7.4 and 7.5.

**Channel benchmarks:** As part of our work in Chapter 5, we benchmarked our improved Flush+Flush primitive, and wrote the `covert_channels_evaluation` and `covert_channels_benchmark` crates to do so. They are described in Section 7.6

**Cache Observer:** As part of our work in Chapter 6, we built a crate to observe the state of the cache evolution in reaction to sequences of memory accesses. This crate, named `CacheObserver` is described in Section 7.7

**Cache study utilities:** A series of utilities used by various parts of the framework. This category comprises the `cache_utiles`, `cpuid`, and `turn_lock` crates. `cache_utils` includes the topology-aware calibration algorithm, described in Section 7.2. The remainder of these utilities is described in Section 7.8.

Additionally, the following external dependencies are used:

- `arrayref`: Used to materialize array references from raw addresses in the bare-metal set-up.
- `bootloader`: Bare-metal bootloader, handing off control to our kernel already in long mode, part of the infrastructure of Oppermann [83].
- `bit_field`: Used by `cache_side_channel` and `covert_channels_evaluation` to access individual bits of integral types.
- `bitvec`: Used in `cache_utils::ip_tool`. This module is used to create copies of functions with specific alignment constraints. The bit vector is used as part of the dedicated allocator.
- `hashbrown`: This is the implementation of `HashMaps` on top of `alloc`, underlying the Rust standard library implementation, but available in free-standing contexts. It is used by `cache_utils`, and whichever `HashMap` implementation is available is re-exported for use by dependent crates.
- `itertools`: Additional methods for iterators, used by `cache_utils` and `CacheObserver`.
- `lazy_static`: Used by the bare-metal to initialize global state in the kernel and drivers.
- `linked_list_allocator`: Allocator used in the *Writing an OS in Rust* blog [83] Operating System, which we also use.
- `libc`: Used by the `cache_utils::frequency` work in progress module to request the CPU frequency from the kernel. It is also a transitive dependency of `nix` below.
- `nix`: Provide idiomatic Rust APIs to the Unix system interfaces. It is used by hosted modules that need to specify the scheduler affinity. This includes `aes-t-tables`, `cache_utils`, `basic_timing_cache_channel`, `cache_side_channel`, `CacheObserver` and `covert_channels_benchmark`.
- `openssl`: Used in `aes-t-tables` to link OpenSSL and invoke it.

- `memmap2`: Used in `aes-t-tables` to map the library file in memory so it can be subjected to a side-channel attack.
- `rand`: Random numbers used in several places.
- `spin`: Used in the bare-metal support to provide proper locking.
- `static_assertions`: Used to prevent any attempt of building with both bare-metal (`no_std`) and hosted (`use_std`) and hosted support in crates such as `cache_utils` where those features are incompatible.
- `volatile`: Used in the drivers' bare-metal support for memory accesses.
- `x86_64`: Used by the bare-metal kernel and driver to access various x86 system mode features. It provides abstractions around page tables, interrupts, and other features used by operating systems. It should not be confused with `core::arch:x86_64`, part of the Rust core library, which provides access to Intel intrinsics.

> **Takeaway:** The framework is built of several modules, called *crates* in the Rust ecosystem, with separate responsibilities, which can be grouped into bare-metal support, cache study utilities, cache channel abstractions, channel benchmarks, and the CacheObserver top-level crate.

We will now cover the framework's main features from the bottom up.

## 7.2. Calibration

One of the key features of the framework is the topology-aware calibration algorithm. Two variants of this algorithm are implemented: `fn calibrate_2t` is a multithreaded version that requires a hosted environment with threads but will measure all possible core pairs of attacker and victim. This version can only run in hosted context, and relies on the `struct TurnHandle<T>` provided by `turn_lock` (see Section 7.8.4). When these assumptions are not verified, `fn calibrate` can be used instead. This function then assumes the victim and the attacker are colocated on the current core. In either case, if the slice hashing function is known, the algorithm calibrates each slice by taking one address in each per-page equivalence class for this function. Otherwise, the algorithm calibrates each cache line separately, which is slower but always correct.

The result of a given combination of an attacker, a victim, and a cache line is a histogram of hit and miss execution times. From these histograms, it is then possible to estimate for a given threshold what the error rate would be. This is how we computed the values in Section 5.4. These histograms are also used by the topology-aware cache primitives described later to set the threshold for each cache line.

For a given histogram, we consider the best threshold to be the one that minimizes the number of misclassifications. We compute the cumulative distribution of hits and misses in linear time and then do a linear search for the thresholds that minimize the misclassified hits and misses in the two hypotheses that hits are faster than misses and that hits are slower than misses. We then select the best of those two candidates.

One limitation of this approach is that it cannot deal with more than two outcomes and also is unable to handle a case where one outcome lies in a range and the other outcomes outside a range, which is what can be observed with dual-socket machines when the attacker position relative to the victim varies. As seen in Section 5.4.3, in such cases, hits with the attacker and victim on different sockets are slower than misses, themselves slower than hits when both the attacker and victim are on the same socket.

> **Takeaway:** The calibration algorithm builds histograms of execution time for hits and misses for each possible combination of an attacker, victim, and cache line, from which the best thresholds for the attacker model are computed. When the slicing functions are available and linear, per cache line calibration is replaced by a calibration once per equivalence class of lines for the slicing functions within each page.

## 7.3. Bare-metal support

The bare-metal support is derived from the x86_64 operating system tutorial from Phillip Opperman [83]. It thus relies on several dependencies that implement most of the requirements to boot an x86 system in long mode, including a bootloader that sets up long-mode paging before handing over control to the 64-bit OS. The bare-metal environment includes a memory allocator (provided by the `linked_list_allocator` crate). It is thus possible to use collections provided by `alloc` and other collections that support `no_std` environment with alloc, which is the case of `hashbrown`, the crate providing the implementation of Rust standard `HashMap`.

Exerting control over interrupts is the remaining specific constraint of the bare-metal support that is not fulfilled by the above. The original tutorial implements conventional drivers for serial ports that rely on preemption. We thus had to re-implement a serial port driver that would not be interrupt-driven. The alternative to being driven by interrupts is to use polling.

Polling is inefficient as it will generally waste resources checking the serial port when no message has been received and will be unable to react immediately to incoming messages. However, in our context, the framework does not expect to receive input while running the experiments and will not be running anything else while expecting input. It is thus acceptable to use polling in this specific case. The implementation of this driver is located in the `polling_serial` crate.

To provide more visible feedback to the experimenter, we also implemented a driver for the firmware-provided character framebuffer, which supports ASCII characters along with background and foreground color. It is found in `vga_buffer`.

> **Takeaway:** The bare-metal support derives from an existing x86_64 Rust operating system tutorial [83] with adaptations to remove interrupt-induced noise, including a polling-based serial port driver.

## 7.4. Channel Abstractions

We developed several abstractions, which we discuss in this section. In most cases, those abstractions correspond in Rust to *Traits*. These can be compared to interfaces or pure virtual classes

in other languages. One major specificity is that Rust has no inheritance of implementations, only of interfaces.

First, there are definitions of an abstract interface to channels similar to Flush+Reload, channels that monitor specific memory addresses. These are described in Section 7.4.1. Then there are specific trait interfaces that correspond to particular use cases, such as side-channel attacks on a table (Section 7.4.2) and covert channels (Section 7.4.3). Last, there is a generic implementation for cache channels that rely on timing a specific instruction. This implementation is described in Section 7.4.4.

*All these abstractions have been designed for the hosted context and are unsuitable for the bare-metal set-up.*

### 7.4.1. The Single Address and the Multiple Addresses Channel traits

The `cache_side_channel` crate defines these two traits. These abstractions were designed so that it would be easy to switch the channel used by a specific piece of code and to allow writing code without regard for the channel used. We defined two distinct traits for different use cases and channel assumptions:

- `trait SingleAddrCacheSideChannel`: For channels monitoring a single cache line at a time.
- `trait MultipleAddrCacheSideChannel`: For channels that support monitoring several lines at a time.

Listings 7.1 and 7.2 shows and documents simplified definition of those two traits.

There are four steps to using those channels:

1. *Calibration*: This creates handles for each address to be targeted. Because we have shown in Chapter 4 that a different threshold may be required for each slice, and given these cannot be predicted ahead of time, the calibration has to be done as part of the initialization of the attacker. Storing the threshold in the handle makes the actual attack step $O(1)$.
2. *Set-up*: The `fn prepare` does the first-time set-up of an address, for instance, flushing it from the cache in Flush+Reload and Flush+Flush attacks.
3. *Running the victim*: The victim is run. This can be code in the process invoking the victim (system call, network request, function call) or simply a suitable delay.
4. *Attack*: This is the phase that tests whether the lines were accessed using the `fn test`. It uses the information in the handle to efficiently run the attack on a given memory address and distinguish outcomes. Using the `reset: bool` argument, it will ensure the locations are then re-set up as if `fn prepare` had called.

Steps 3. and 4. can be repeated as many times as needed. If the reset option is not used, 2. must also be repeated between measurements. In covert channel contexts, the victim is the transmitter, and the attacker is the receiver.

> **Takeaway:** The `trait MultipleAddrCacheSideChannel` and `trait SingleAddrCacheSideChannel` abstractions make it easy to write cache side or covert channel code without regard for the underlying primitive used.

```rust
pub trait MultipleAddrCacheSideChannel: /* ... */ {
    type Handle: ChannelHandle; // wraps the address and metadata required by a channel

    // Maximum number of addresses that can be monitored in one go, 0 for no limit
    const MAX_ADDR: u32;

    // monitor up to MAX_ADDR lines from the Vec, optionally reset after the measure.
    unsafe fn test<'a, 'b, 'c>(
        &'a mut self,
        addresses: &'b mut Vec<&'c mut Self::Handle>,
        reset: bool,
    ) -> Result<Vec<(*const u8, CacheStatus)>, SideChannelError>
    where
        Self::Handle: 'c;

    // do any setup required for the lines up to MAX_ADDR lines from the Vec.
    unsafe fn prepare<'a, 'b, 'c>(
        &'a mut self,
        addresses: &'b mut Vec<&'c mut Self::Handle>,
    ) -> Result<(), SideChannelError>
    where
        Self::Handle: 'c;

    // execute an operation in the victim context, in PoCs
    fn victim(&mut self, operation: &dyn Fn());

    // take addresses and run the calibration required by the channel, returns the Handles
    ↪ used by the other methods
    unsafe fn calibrate(
        &mut self,
        addresses: impl IntoIterator<Item = *const u8> + Clone,
    ) -> Result<Vec<Self::Handle>, ChannelFatalError>;

    /// # Safety
    /// The address passed must be valid to read
}
```

Listing 7.1.: Cache Side Channel Traits

```rust
pub trait SingleAddrCacheSideChannel: /* ... */ {
    type Handle: ChannelHandle;
    unsafe fn test_single( &mut self,
        &mut Self::Handle, bool,
    ) -> Result<CacheStatus, SideChannelError>;
    unsafe fn prepare_single( &mut self,
        &mut Self::Handle
    ) -> Result<(), SideChannelError>;
    fn victim_single(&mut self, &dyn Fn());
    unsafe fn calibrate_single( &mut self,
        impl IntoIterator<Item = *const u8> + Clone,
    ) -> Result<Vec<Self::Handle>, ChannelFatalError>;
}
```

Listing 7.2.: Single Address Cache Side Channel Trait, similar to the Multiple Address trait in Listing 7.1, but optimized for monitoring a single address.

### 7.4.2. The Table Side Channel trait

In some situations, it is possible to simplify the interface. When running a side-channel attack where one monitors memory accesses to a fixed set of memory addresses, such as a table of read-only data, the attack can be simplified to require the interface shown in Listing 7.3. This interface can be implemented on top of the previous traits or directly.

> **Takeaway:** The `trait TableCacheSideChannel` simplifies writing side channel attacks monitoring the set of accesses to shared memory by victim code.

### 7.4.3. The Covert Channel trait

Similarly, in covert-channel contexts, the interface suitable for a primitive can be simplified to the `trait CovertChannel`, presented in Listing 7.4. It assumed the channel could use one or more pages and have one handle per page. Channels may transmit several bits on a single page. The synchronization protocol is left to the user of the channel.

> **Takeaway:** The `trait CovertChannel` simplifies the use of covert channels by abstracting away most details.

### 7.4.4. The Primitive trait and the generic implementations

We found that the code we wrote for Flush+Flush and Flush+Reload were strikingly similar. The main differences between the two implementations of the above traits were that Flush+Flush did not need a reset after taking measurements and the exact instruction that would be timed.

Consequently, we wrote a generic code that only requires specifying the measurement function, returning a numeric ”time” value (which could, in fact, represent a different measure than a time). The generic implementation then turns this primitive into a fully usable cache channel, providing implementations of all four cache channel traits.

```rust
pub trait TableCacheSideChannel<Handle: ChannelHandle>: /* ... */ {
    // Determine threshold if needed and create handles
    unsafe fn tcalibrate(
        &mut self,
        addresses: impl IntoIterator<Item = *const u8> + Clone,
    ) -> Result<Vec<Handle>, ChannelFatalError>;

    // measure num_iteration times the victim impact.
    unsafe fn attack<'a, 'b, 'c, 'd>(
        &'a mut self,
        addresses: &'b mut Vec<&'c mut Handle>,
        victim: &'d dyn Fn(),
        num_iteration: u32,
    ) -> Result<Vec<TableAttackResult>, ChannelFatalError>
    where
        Handle: 'c;

    /// # Safety
    /// addresses must contain only valid pointers to read.
}
```

Listing 7.3.: The `trait TableCacheSideChannel`, meant to simplify running side channel attacks on tables

```rust
pub trait CovertChannel: /* ... */ {
    type Handle; // Contains metadata, per page.
    const BIT_PER_PAGE: usize; // How many bits can be transmitted using a page

    // encode data into the channel on a given page,
    // BitIterator wraps a Vec<u8> to extract individual bits.
    unsafe fn transmit(&self, handle: &mut Self::Handle, bits: &mut BitIterator);

    // receives data and returns the bits received in a page in one transmission round.
    unsafe fn receive(&self, handle: &mut Self::Handle) -> Vec<bool>;

    // set-up the page before it can be used to transmit.
    unsafe fn ready_page(&mut self, page: *const u8) -> Result<Self::Handle, ()>;
}
```

Listing 7.4.: The `trait CovertChannel`, meant to simplify writing covert channels

```rust
pub trait TimingChannelPrimitives: /* ... */ {
    unsafe fn attack(&self, *const u8) -> u64;
    const NEED_RESET: bool;
}
```

Listing 7.5.: Simplified definition of the Timing Primitive Trait

This will guarantee that the results of different primitives are pretty comparable, and it further automatizes Flush+Flush and Flush+Reload channels, as one no longer needs to pick the threshold by hand.

The `basic_timing_cache_channel` crate thus provides a generic implementation of both (`trait MultipleAddrCacheSideChannel` and `trait SingleAddrCacheSideChannel`) for such channels. It is parameterized by a type implementing the `TimingChannelPrimitives` *trait* (a Rust interface), defined in Listing 7.5.

The `attack` method returns how long it took for the attack sequence operation to execute, or more generally, a numeric result where hit and miss distribution can be distinguished. The `NEED_RESET` constant indicates whether an extra `clflush` is required to reset the line after the attack. For instance, this is needed for Flush+Reload, but not for Flush+Flush.

We also include a *naive* version of the generic timing channel, which uses a global threshold specified by the user instead of the calibration routine. This is an efficient way of comparing the topology-aware primitives above with the usual, simpler, topology-unaware implementations.

One limitation of the current design is that it assumes the set-up state for cache lines is the Invalid coherence state. The implementation could be extended to include a reset function in addition to the attack function. However, it enabled us to mutualize the full implementation of Flush+Flush and Flush+Reload, and reduce the `flush_flush` and `flush_reload` crates to a minimal amount of code.

> **Takeaway:** `basic_timing_cache_channel` crate provides a generic implementation of a topology-aware and naive cache attack of the Flush+X family, where X is an operation returning a numeric value exhibiting distinct hit and miss distributions. It guarantees our Flush+Reload and Flush+Flush implementations are comparable.

## 7.5. Implementations of Flush+Reload and Flush+Flush

We split the implementation of each cache channel into a separate crate to avoid pulling all of them in needlessly. The `flush_flush` and `flush_reload` crates are structured identically, with the top-level module providing the topology-aware primitive and the `naive` submodule providing the naive version of the primitive, which uses a global threshold.

Listing 7.6 presents the sequences of instructions used for measurement. The `fn only_flush` is used for the `flush_flush` crate, the similar `fn only_reload` is used for `flush_reload`.

```rust
pub unsafe fn rdtsc_fence() -> u64 {
    unsafe { core::arch::x86_64::_mm_mfence() };
    let tsc = unsafe { core::arch::x86_64::_rdtsc() };
    unsafe { core::arch::x86_64::_mm_mfence() };
    tsc
}
pub unsafe fn only_flush(p: *const u8) -> u64 {
    let t = unsafe { rdtsc_fence() };
    unsafe { core::arch::x86_64::_mm_clflush(p) };
    // OR core::ptr::read_volatile(p) for only_relaod
    (unsafe { rdtsc_fence() } - t)
}
```

Listing 7.6.: Measurement primitives

> **Takeaway:** The `flush_flush` and `flush_reload` crates provide implementation of both topology-aware and naive global-threshold Flush+Flush and Flush+Reload, using the `basic_timing_cache_channel` generic implementation.

## 7.6. Channel benchmarks

In addition to the aforementioned error estimation, we implemented two programs that use our abstractions and measure the performance of different channels.

The abstractions in Sections 7.4.2 and 7.4.3 are the foundation on which each benchmark is built. Each benchmark has a library that defines the benchmark relying only on the interface and a main program that instantiates it with specific implementations of the abstractions. Only the main program depends on the `flush_reload` and `flush_flush` crates.

### 7.6.1. Covert Channel Benchmark

The first of these benchmarks is the covert channel, where we transmit bits through a channel. To really measure the bandwidth of the channel itself, we leverage an ideal synchronization primitive, the `struct TurnHandle<T>` provided by `turn_lock` (see Section 7.8.4 later).

There are two components to this feature. First, a library crate, `covert_channel_evaluation`, defines a general benchmark, which only relies on the cache channel interfaces. Then a binary crate called `covert_channel_benchmark` instantiates the benchmark with the various covert channels (`type FlushAndFlush` and `type NaiveFlushAndFlush` for `flush_flush`, and `type FlushAndReload` from `flush_reload`).

#### The covert channel evaluation harness

Relying on the previously discussed `trait CovertChannel` (Section 7.4.3), we measure the performance of different channel primitives using an ideal synchronization primitive. The benchmark is parametrized with the number of bytes transmitted and the number of pages used. We use two threads: a transmit thread and a receive thread. Each page is guarded using a

separate turn lock. The transmit and receive threads cycle through the pages until the correct number of bytes has been received. The transmit thread generates the requested number of random bytes and returns the original vector after terminating. The main thread runs the receiver, and after receiving all the bytes, it joins the transmitter and compares the results. This allows the benchmark to compute the number of errors and return it.

We also measure the execution time between the start of the transmission (in the transmit thread) and the end of the reception in the receiver thread, both using the standard library call to obtain the time in seconds and using `rdtsc` to get the time in cycles.

Overall, for a given channel and numbers of bytes and pages, we thus return a structure `struct CovertChannelBenchmarkResult` containing the number of bytes transmitted, the number of errors, the error rate, and the benchmark run time in both seconds and cycles.

#### Instantiation of the benchmark

The `covert_channel_benchmark` binary crate depends on the previous crate along with the `flush_flush` and `flush_reload` channel-implementation crates. It instantiates the four channels, both naive and topology-aware Flush+Flush and Flush+Reload. It runs the benchmark `NUM_ITER = 16` times for each channel and each page number between 1 and 32. In addition, for a naive channel, the benchmark is iterated 16 times per core pair to ensure proper coverage of the variability caused by the topology. This makes the experimental results more reproducible and better reflect reality. The topology-aware channel selects an address belonging to the best possible slice, whereas the naive channel always picks the first cache line of the page.

We then compute the average and variance of the bandwidth, error rate, and true capacity, which can be plotted to identify the optimal configuration for each channel. This is thus the algorithm that underpins our results in Section 5.5.1. Its main limitation is the assumption that each page is independent. Such an assumption does not make sense for a Prime+Probe-based covert channel, for instance, as such a channel relies on cache set contention, which is not independent across different pages.

Those two crates require relatively little code, thanks to the previously written abstractions (around 225 lines each).

> **Takeaway:** The two crates `cover_channel_evaluation` and `covert_channel_benchmark` provided us with a good quality benchmark of the capacities of various cache-based covert channels, whose result we showed in Section 5.5.1.

### 7.6.2. AES Side channel

To evaluate the channels in a side-channel context, we reproduced the AES chosen plaintext attack from [37], which targets a well-known vulnerable implementation in OpenSSL, usually disabled by default. The `aes-t-tables` crate implements the attack. However, it requires a vulnerable OpenSSL library, which has to be manually built. We include a wrapper script, `./cargo.sh` around the Rust build-system that can be used to set the correct environment to use the vulnerable library. The `OPENSSL_DIR` environment variable in the script should be set to the path of the directory where the vulnerable OpenSSL has been built. Given this program is a

proof of concept, the program will make calls to OpenSSL's `aes_ige` with the chosen plaintexts and the key while measuring the side channel and returning the attack results. One can then check if those results reflect the key used.

The crate is divided into a library and a main program. The library provides a single entry point, `unsafe fn attack_t_tables_poc(...)` which runs the attack. This function requires the channel to be used along with a parameter struct containing the path to the targetted library, the number of iterations of the attack, the key, and the addresses of the targeted tables. The main program is then responsible for creating the three channels (topology-aware and naive Flush+Flush and the usual Flush+Reload) and selecting the correct parameters. Currently, one needs to edit the program to provide the addresses for the T-tables for the OpenSSL library built.

The attack is written using the `trait TableCacheSideChannel` and is a rather straightforward application of our abstractions (less than 250 lines of code). For each iteration, it generates a set of random plaintext bytes to complement the one byte that is kept fixed. It flushes the T-tables out of memory, runs the victim operation, and finally measures which elements of the table have been brought back into the cache. It will run `num_encryptions` for each value of the byte considered (byte 0 by default).

Plotting a heatmap of the line accessed depending on the value of the fixed byte will show certain lines are deterministically accessed while others are accessed more randomly. The pattern formed by those lines accessed deterministically is correlated with part of the key. Repeating the attack for every single byte of the plaintext reveals half of the bits of the key. From there, it is usually possible to break the encryption. However, given our focus is on the covert channels, we only focus on the quality of the heatmap obtained and do not implement the result exploitation part.

> **Takeaway:** The attack on the OpenSSL implementation of AES using T-tables is a good application of our side channel abstractions.

## 7.7. Cache Observer

The core of the framework is the library providing the `struct Prober` object, which can be used to repeatedly run a pattern (sequence of memory accesses in a page or a group of several consecutive pages) and measure the result.

A probe pattern is defined as a `Vec<PatternAccess>`, where `PatternAccess` is a structure containing an offset and an access function taking a pointer to a byte and returning a 64-bit value. It is designed to control the exact instruction doing the memory access, leveraging the `ip_tool` module described later (Section 7.8.2). It also permits collecting data about how long the various access in the pattern took, in addition to the final state of each line.

The `struct Prober` object is instantiated with two parameters, a page group size ($G$) and a number of groups. However, in most of our experiments, the groups are of one page, and we thus generally refer to them as pages instead of page groups. The object then allocates `num_pages` range of $G$ pages using `mmap` and calibrates them for the Flush+Flush and Flush+Reload channels. These two channels permit implementing three strategies to measure the pattern impact, which were defined as follows in Section 6.2:

1. `SingleFlush`: In this strategy, a single address is probed with `clflush` after a run of the pattern, and 64 times more iterations are required to cover the full page.
2. `SingleReload`: This is the same strategy using reload operations as the measurement primitive. It is the only strategy usable with loads, as more measurement loads would interfere with the prefetcher operation. This is the baseline approach used in the literature.
3. `FullFlush`: This is the most efficient technique that flushes the whole page(s) after the pattern has been run.

The `fn full_page_probe` will run a pattern a specific number of iterations, after a specified number of warm-up iterations with the three strategies, in the following order `SingleFlush`, `SingleReload`, and `FullFlush`, and return a structure containing the results. Notably, the result can also include information returned by the function used to make the accesses, such as how long individual accesses took. This timing is currently implemented using `rdtsc` and fences, which is the reason why L1-prefetchers were not observed, as mentioned in Section 6.2.1. However, the functions used in the pattern are set by the user and can thus be easily changed.

The various experiment binaries generate large sets of patterns, instantiate the `struct Prober`, and then iterate over the set of patterns invoking for each of them `fn full_page_probe` and outputting the results.

> **Takeaway:** The `struct Prober` object is the core of the `CacheObserver` framework, which underpins Chapter 6. It can be used to obtain the final state of the cache after a sequence of memory access, which may be useful for purposes other than simply reverse engineering.

## 7.8. Other Utilities

Several smaller utility modules have been mentioned previously, which we will describe here.

### 7.8.1. Anonymous memory map support

This module, `cache_utils::mmap`, was written to provide a safe wrapper around `mmap` to obtain pages with specific configurations. Typically this is used to obtain 2 MiB huge pages or writable and executable (W+X) pages. Huge pages require a specific option that was not exposed by the existing crates when this code was written. The pages can be presented as an array of a `type T` chosen at the creation of the pages. The `struct MMap<T>` exposes several constructors to control how to initialize the page. Each object exposes the pages it owns as an array slice. Pages are unmapped automatically when the owning `struct MMap<T>` is destroyed.

> **Takeaway:** `cache_utils::mmap` provides a safe abstraction around `mmap` exposing specific options required by our research, huge-pages and W+X pages.

### 7.8.2. IP-tool

This module is initially written as part of the `CacheObserver` framework, aka `prefetch_reverse` in earlier framework versions. It allows the instantiation of a function template with specific

alignment constraints. In practice, the functions must have a simple signature and obey the "C" ABI. In addition, the template must be able to provide symbols at the start and the end of the function. Additionally, one symbol should point to the instruction whose alignment will be controlled.

It was later refactored into `cache_utils` as it could be used for other purposes, for instance, to study attacks involving the instruction caches.

### 7.8.3. `cpuid` and slices

To run the appropriate code, the targeted microarchitecture must be identified. To automate this in our framework, we built a crate, based on the tables in Intel's Manual [47], which uses the `cpuid` instruction to query the processor manufacturer (leaf `0x0`), and then the model, family, and stepping, of the CPUs (leaf `0x1.EAX`). From there, the crate deduces the CPU's microarchitecture. This crate only supports microarchitecture detection on Intel CPUs. It detects the AMD CPUs but cannot identify their microarchitectures.

This knowledge is then used in the `cache_utils::complex_addressing` module to detect whether the hash function used by the sliced L3 is known. The calibration algorithm then selects an optimized strategy when the slicing functions are known. In that case, the calibration is run once per slice and page instead of once per cache line.

### 7.8.4. Turn Lock

This crate is a synchronization primitive that helps coordinate two or more threads to ensure a specific order of operation issued from different cores. Each thread receives a `struct TurnHandle<T>`, with the number of handles being set at the creation of the lock using `fn new(num_turns: usize, data: T) -> Vec<TurnHandle<T>>`. Each thread calls `fn wait(&mut self) -> TurnLockGuard<T>` to wait for its turn. The guard returned allows access to the shared data. Releasing the lock destroys the guard and assigns the lock to the next thread. This lock enforces that each thread gets the lock at its turn only. When the lock is assigned to a thread, this thread may not be waiting already, but `fn wait` will return immediately when the thread calls it. Obviously, if a thread fails to acquire or release the lock, it will stall every other thread. Ensuring proper termination is subtle.

Implementation-wise, this primitive is meant for the turn to be fast and `fn wait` spins while waiting. Each handle contains its handle number and a reference to the internal shared structure, which contains both the number of handles and an atomic value indicating the current holder. One enhancement would be moving from a test-and-set approach to a test and test-and-set approach, which would improve the performance.

### 7.8.5. Analysis

The previous experiments generated several GiB of data. More precisely, the various experiments output log files that embed one or more CSV result files. In most cases, we use ad-hoc shell scripts to extract those CSVs, which are then read by Python or Julia scripts to generate various plots. The `analysis` folder contains the scripts used by the `CacheObserver` experiments.

The other experiments' scripts can be found in the respective crates' folders. Additional work could be undertaken to make the analysis infrastructure more robust and easier to use.

## 7.9. Discussion

### 7.9.1. Advantages

First, using a modern programming language has facilitated the development of the bare-metal solution and of powerful abstractions, making it easy to apply the cache channel primitives to various problems. The value of tools such as generics, traits, and advanced data structures combined with zero-cost abstractions, predictable memory management, and the availability of low-level operations cannot be understated for this work.

Thus, the calibration algorithm and the generic implementation make building robust and automated primitives easier and reduce the work needed. In addition, the framework has shown flexibility and can be used in many cases or easily extended to support new use cases. While most of the primitives currently assume the presence of threads, it should be easy to add a configuration option to remove this requirement and use a single-threaded calibration in environments such as bare-metal.

The CacheObserver crate can be used for many different purposes, not only prefetcher reverse engineering. Its current API makes access patterns pretty flexible, supporting many use cases.

### 7.9.2. Limitations

First, our choice of programming language does come with certain drawbacks. The Rust ownership model and type system may require careful design to make the architecture satisfy the borrow-checker. It can also make certain parts of the API heavy to use to deal with memory ownership properly. This is for instance the case in the CacheObserver experimental programs which feature types such as `Box<dyn Fn(usize) -> (usize, usize, Box<dyn Fn(usize, usize) -> Vec<usize>>)>` to represent closures returning a closure in a way compatible with storing several of them inside a collection; or in the API of the `trait MultipleAddrCacheSideChannel` which uses a `&'b mut Vec<&'c mut Self::Handle>` parameter to satisfy the constraint of the borrow-checker.

Some of Rust's low-level features we require, especially those involved in bare-metal support, are still deemed unstable and require the use of the nightly version of the compiler, with the possibility of breakage at each compiler update. We document the known-to-work compiler versions for this reason.

Second, The bare-metal support is also not well integrated within the rest of the framework, chiefly because it cannot currently boot several cores and run multiple threads. Either feature would be possible, provided development work.

There are also portability issues. For AMD CPUs, several lower-level utilities, such as `cpuid`, are missing the requisite information. Adding this information and some debugging will be necessary to port the framework on those platforms. Once this is done, the Flush+Flush channel and the CacheObserver framework may still not work, as it is currently unknown whether those machines are vulnerable to Flush+Flush.

There are also strong dependencies on x86, especially the assumption of a `clflush` or equivalent instruction. This may make porting the framework on ARM challenging. ARMv8 includes suitable instructions in the `DC` family of instructions, but it may require integration with the operating system to enable it in user mode (EL0) or grant sufficient privileges to the framework (EL1). The latter is possible for reverse engineering operations but not realistic when building attacks. The channel abstractions would work using the techniques proposed by Lipp et al. [59], provided sufficient development. However, the dependency of the CacheObserver framework on a cache primitive such as Flush+Flush means it may not be possible to port it to ARMv8. It is currently unknown whether the `DC` instructions present different execution times for hits and misses.

More generally, this framework was designed with Flush+Reload and other shared-memory stateful cache attacks in mind. The design is less adapted to primitives, such as Prime+Probe, that exploit cache set conflicts with the victim or transient attacks.

On the covert channel side of things, the framework does not include proper synchronization for realistic settings. Similarly, its side channel modules do not necessarily include the tools to set up attacks, such as identifying targets in existing libraries, such as done in [37].

> **Takeaway:** Using the Rust programming language for reverse engineering and microarchitecture security research has many advantages but also has a few drawbacks. Most of these issues can be worked around provided a good familiarity with the language, which is thus a viable and valuable tool for such research. Our framework has significant flexibility when applied to Intel CPUs and using Flush+Reload-like primitives but currently suffers from limitations when dealing with different CPUs or primitives that do not operate on shared memory.

## 7.10. Summary

In this chapter, we have looked at the design and implementation of our research framework, regrouping un the name *dendrobates* the code for the research published in our *Calibration Done Right: Noiseless Flush+Flush Attacks* and *Characterizing Prefetchers using CacheObserver* (Chapters 5 and 6) with additional experiments, presented in Chapter 4. We first looked at the various design goals, our choice of programming language, and the overall architecture. Then we developed the different parts of the implementations, providing additional details compared to the publications reproduced in the previous chapters. Lastly, we discussed the merits and issues of our framework.

# Conclusion

In this chapter, we can now review the answer to our original research questions and then reflect on the perspectives opened by this research for future work.

With our first publication, *Calibration done Right: Noiseless Flush+Flush* [26], we concluded that the interconnect topology is responsible for `clflush` timing variation and modelled these variations (**Q3**). We also showed how a topology-aware calibration algorithm could improve Flush+Flush to be a primitive as reliable as Flush+Reload, and provide a better channel capacity (**Q4**).

Building upon this work, our second paper *Characterizing Prefetchers using CacheObserver* shows how Flush+Flush is a primitive suitable to reverse engineer parts of the memory caches such as prefetchers. Thanks to the ability to control core placement in this setting, it provides excellent accuracy, while it does not interact with prefetcher the same way Flush+Reload does (**Q2**). Thanks to this improved Flush+Flush, it is thus practical to monitor a range of memory for the activity of prefetchers, which provides significantly more data and can bring to light prefetches in unexpected places. Thus, this approach is viable for prefetcher reverse engineering (**Q1**).

Overall, this research has developed a powerful tool and method to study the behavior of prefetchers that enables monitoring their activity within a range of memory addresses, building on an improved Flush+Flush cache side channel.

## Future perspectives

Building from there, there are several directions for the future. We will first look at the next potential steps in prefetcher reverse engineering, then consider the larger applicability of both Flush+Flush and the techniques developed in this thesis.

### Prefetcher reverse engineering

Our framework is similar in its ability to the CacheQuery tool built by Vila etal. [122, 123] as part of their framework to reverse engineer eviction policies. Their framework layered higher levels of abstractions on top of CacheQuery. This layering allowed a learning algorithm to learn the actual policies, while CacheQuery received sequences of memory accesses to run and gave the result of these. Our CacheObserver framework could thus be used as the foundation upon which build a framework to infer how prefetchers behave. Building such higher-level abstractions is the first serious continuation of our work that could be done. Challenges may include the non-determinism of prefetchers, which we have observed in several instances. The behavior space of prefetchers is also significantly larger than that of cache sets. Identifying what lines the prefetcher is susceptible to fetch may be needed to reduce the search space sufficiently.

The current framework has been limited to L2 prefetcher by its rather aggressive use of fences while executing access patterns. As mentioned by the Intel optimization manual [46], this inhibits the L1 prefetchers. Therefore, it would be valuable to develop an alternative technique to ensure proper ordering of the various memory accesses, also allowing the prefetcher to observe L1 prefetchers.

Once this issue is fixed, it would also enable research into the interaction of the various prefetchers, which our research has observed between the two L2 prefetchers in Intel CPUs. Furthermore, with both cache levels supported, it would notably allow us to determine if the prefetches issued by the L1 caches are treated at the L2 level differently from demand misses or not.

This research direction might also benefit from gaining the ability to identify the cache level a line has been fetched to, which could require the calibration algorithm to be adapted to categorize execution times into several categories instead of two. For instance, classifying a load as a load from Memory, L3, L2, L1, instead of simply as a hit or a miss.

Altogether, a great deal of further research is required to understand in detail the collective effects of the various prefetchers in these CPUs.

> **Takeaway:** The natural continuation of our work is to build higher levels of abstractions on top of the Cache Observer framework in order to apply techniques similar to Vila's reverse engineering of eviction policies.

### Larger applicability

Our research has used a pair of Intel CPUs with closely related microarchitectures, as Coffee Lake and Whiskey Lake are both minor revisions of the Skylake architecture; it is thus worth considering the wider applicability of these techniques. First of all, given the interconnect layout was relatively conserved since its introduction in Sandy bridge, along with the disclosed prefetcher, our contribution likely applies to all earlier CPUs until Sandy Bridge. However, little further study has been applied to Flush+Flush on other microarchitectures.

More recent CPUs introduced non-inclusive caches; however, because `clflush` applies to the entire coherent domain, its execution time may be affected. Given what we have seen, the `clflush` instruction requires communication over the interconnect, depending on which core may contain a copy in a private cache. The communication likely differs depending on the cache coherence state. Unless manufacturers purposefully ensure `clflush` executes in a constant time, it seems likely that differences in execution time may remain observable. Surprisingly, no publication has been made about `clflush` execution times on AMD CPUs. Surveying the different microarchitectures of x86 CPUs for their vulnerability to this kind of attack would be low-hanging fruit. We estimate it is quite likely that `clflush` remains a valid channel on contemporary architectures.

In addition, ARM v8 introduced the `DC` data cache maintenance family of system instructions [7], which may be available in EL0, the unprivileged mode of execution on AMR. However, it is currently unclear whether these instructions are implemented in constant time and which operating systems enable them in user mode (EL0 in ARM terminology). If this instruction is usually disabled in user mode, it may not be used to mount cache attacks but would not prevent

its use for reverse engineering purposes.

Additionally, the calibration algorithm may be applicable for other cases where timing differences induced by cache coherence depend on interconnect topology and cache organization. As CPUs grow in core number, the impact of topology and cache organization on execution is likely to increase, making such an algorithm increasingly important to build accurate channels.

The Cache Observer overall approach may also remain applicable as long as a cache channel that does not trigger prefetcher operation can be found. Any instruction that is not deemed to be a memory access by the prefetcher but has a different execution time depending on whether a line is cached may constitute such a primitive.

> **Takeaway:** A second extension of our work would be to evaluate the applicability of the Flush+Flush primitive to a variety of microarchitectures. Overall the topology-aware algorithm and the Cache Observer approach are likely to remain applicable widely, possibly using other instructions.

# Appendices

# A. Reverse Engineering the Intel Slice Function

Our research relies on having prior knowledge of the cache slicing functions. We have updated the code base used by Maurice et al. [67] to support newer architectures and used it to uncover the slicing functions of the i9-9900 (Coffee Lake R, 8 cores) and the older i7-4980HQ (Crystal Well, 4-core Haswell with an eDRAM L4 cache), which differ from the previously known functions (see Table A.1) that applied to most CPUs from Sandy Bridge to Broadwell. The CPU in our 4-core machine also uses those well-known functions. The available memory limits the most significant bits that can be uncovered in the function.

This method uses performance counters located in a per physical core structure called CBox. The uncovered functions map addresses onto each CBox. However, it is suspected that starting with Skylake, there are two slices within the same CBox [124], which we cannot detect with this method.

The performance counters we used are located within the CBox, which corresponds to the interconnect node. The performance counters within CBox $n$ are controlled by a set of consecutive MSR whose names start with `MSR_UNC_CBO_`$n$`_`. These MSRs were introduced in CPUs based on the Sandy Bridge microarchitecture.

One difficulty on the Coffee Lake machine is that the *Intel 64 and IA-32 Architectures Software Developer's Manual* [47] only documents the required MSRs for the CBox in core 0 to 4. Trying to infer the correct MSR for the additional CBox, it appears that the eighth MSR (for core 7) would collide with another documented MSR, architectural MSR `0x770`, named `IA32_PM_ENABLE`[1]. We found the other MSRs for core 5 and 6 and adapted the code to exploit the incomplete set of MSRs, with one MSR missing. We document in Table A.2 the performance counters we uncovered.

[1] In the Cannon Lake architecture, Intel reorganized the `MSR_UNC_CBO_`$n$`_` MSRs so that twice as many counters could fit in the same range, and Alder Lake moved those MSR to an entirely different area of the MSR address space, with room for significant growth in core numbers.

Table A.1.: Functions from [67] for the 2-, 4- and 8-core Xeon and Core CPU and new functions for the Intel Core i7-4980HQ and i9-9900. ⊕ represents the exclusive OR operation.

| | | Address Bit | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 |
| Sandy Bridge & later [67] | $o_0$ | | ⊕ | ⊕ | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ |
| | $o_1$ | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | ⊕ | |
| | $o_2$ | ⊕ | ⊕ | ⊕ | ⊕ | | | ⊕ | ⊕ | | | ⊕ | ⊕ | | | ⊕ | ⊕ | | | ⊕ | | | ⊕ | | | ⊕ | ⊕ | | | ⊕ | | | |
| (New) i7-4980HQ | $o_0$ | | | | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | |
| | $o_1$ | | | | | ⊕ | | ⊕ | | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | |
| (New) i9-9900 | $o_0$ | | | | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | | ⊕ | | | | ⊕ |
| | $o_1$ | | | | | | ⊕ | ⊕ | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | ⊕ | ⊕ | ⊕ | | ⊕ | | ⊕ | ⊕ | | | | ⊕ | | | ⊕ | | | |
| | $o_2$ | | | | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | | | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | ⊕ | | | ⊕ | | ⊕ | ⊕ | | | | | |

Table A.2.: Performance counters for CBox 0 to 6 on the Coffee Lake microarchitecture

| Address | Name | Description | Source |
|---|---|---|---|
| 0x700-0x709 | MSR_UNC_CBO_0_xxx | Uncore C-Box 0 Performance Counters | [47] |
| 0x710-0x719 | MSR_UNC_CBO_1_xxx | Uncore C-Box 1 Performance Counters | |
| 0x720-0x729 | MSR_UNC_CBO_2_xxx | Uncore C-Box 2 Performance Counters | Table 2-22, |
| 0x730-0x739 | MSR_UNC_CBO_3_xxx | Uncore C-Box 3 Performance Counters | |
| 0x740-0x749 | MSR_UNC_CBO_4_xxx | Uncore C-Box 4 Performance Counters | Vol. 4 2-195 |
| 0x750-0x759 | MSR_UNC_CBO_5_xxx | Uncore C-Box 5 Performance Counters | (New) Reverse |
| 0x760-0x769 | MSR_UNC_CBO_6_xxx | Uncore C-Box 6 Performance Counters | engineering |

# B. Blueberry pie recipe

## B.1. Sweet Shortcrust pastry *(Pâte sablée)*

**Ingredients:** (1.5 – 2 pie crusts)

- 100g of sugar
- 200g of butter
- 400g of flour
- 1 egg
- *(optional) vanilla sugar or extract*

**Instructions:**

1. Cut the butter into small pieces, add sugar and flour (and add the vanilla sugar if applicable).
2. Work the mix with your finger until you get a sand like texture.
3. Beat the egg and add it to the mix (add a bit of water if needed), form a ball, and put in the fridge for half an hour (under plastic film) (add the vanilla extract if applicable).
4. Spread it in your pie pan.

**Baking:** 25 min at 210 ℃ (483.15 K), then lower the temperature to 180℃ for 15 minutes, or until the fruit are nicely cooked, without burning the crust.

*You may need to lower even further to 150℃ in some cases. Depending on the pie content the crust may be cooked before filling it, e.g., strawberries.*

## B.2. Pastry cream *(Crème pâtissière)*

**Ingredients:**

- 2 eggs yolks and 1 whole egg (or 3 egg yolks)
- 75 to 100g of sugar
- 500mL of milk
- 40g of corn starch (or plain flour)
- *(optional) vanilla sugar or extract, cooking alcohol (e.g., rhum)*

**Instructions:**

1. Put your milk to heat.
2. Meanwhile stif the egg and sugar together energetically (until the mixture color turns lighter).
3. Add the corn starch, and any extra ingredients.
4. Slowly pour the boiling milk onto the the mix while stirring.
5. Put back the mix to heat while continuously stirring until the mix thickens (theoretically stop at the first sign of ebullition).

## B.3. Blueberry pie instructions

**Ingredients:**

- Sweet Shortcrust pastry (see Appendix B.1)
- Pastry Cream (see Appendix B.2)
- 400g of small blueberries
- *(optional) Vanilla sugar*

**Instructions:**

1. Make a Sweet Shortcrust Pastry and spread it in the pan
2. Make the Pastry Cream and pour it in the crust (should fill roughly half the height)
3. Add the blue berry on top of the cream (the cream will absorb the juice (while the berries cook), and take a nice blue color and taste, and will prevent the juice from soaking the crust and making it loose it's mechanical properties.)
4. Add three spoonful of sugar of top of the fruits
5. Bake in the oven as per the cooking instructions for the pie crust.
6. *Sprinkle a paquet of vanilla sugar on top.*

# Bibliography


[1] Onur Aciiçmez and Çetin Kaya Koç. "Trace-Driven Cache Attacks on AES (Short Paper)". In: *Information and Communications Security, ICICS*. 2006.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: *S&P*. 2019.

[3] Thomas Anderson and Michael Dahlin. *Operating Systems: Principle & Practice*. 2014. ISBN: 978-0-9856735-2-9.

[4] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Fine Grain Cross-VM Attacks on Xen and VMware". In: *International Conference on Big Data and Cloud Computing, BDCloud*. 2014.

[5] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a Minute! A fast, Cross-VM Attack on AES". In: *RAID*. 2014.

[6] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily Pao Looi, Sreenivas Mandava, Andy Rudoff, Ian M. Steiner, Bob Valentine, Geetha Vedaraman, and Sujal Vora. "Cascade Lake: Next Generation Intel Xeon Scalable Processor". In: *IEEE Micro* (2019).

[7] *Arm Architecture Reference Manual. Armv8, for Armv8-A architecture profile*. Version DDI 0487F.b (ID040120). ARM Ltd. 2020. URL: `https://developer.arm.com/documentation/ddi0487/latest`.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. Arpaci-Dusseau Books, 2018.

[9] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Communications in Computer and Information Science*. 2013.

[10] Laszlo A. Bélády. "A Study of Replacement Algorithms for Virtual-Storage Computer". In: *IBM System Journal* (1966).

[11] Daniel J Bernstein. *Cache-timing attacks on AES*. Tech. rep. 2005.

[12] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. "Branch Prediction Attack on Blinded Scalar Multiplication". In: *IEEE Transactions on Computers* (2020).

[13] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. "The gem5 simulator". In: *SIGARCH Computer Architecture News* (2011).

[14] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. "Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs". In: *CT-RSA*. 2010.

[15] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks". In: *USENIX Security*. 2020.

[16] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd edition. Pearson Education, 2016. ISBN: 978-0-13-409266-9.

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX Security*. 2019.

[18] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *CCS*. 2019.

[19] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. "Leaking Control Flow Information via the Hardware Prefetcher". In: *arXiv:2109.00474* (2021).

[20] Hong-Tai Chou and David J. DeWitt. "An Evaluation of Buffer Management Strategies for Relational Database Systems". In: *VLDB*. 1985.

[21] *Coffee Lake - Microarchitectures - Intel - WikiChip*. Last edited 2020-07-03. WikiChip LLC, 2020. URL: `https://en.wikichip.org/w/index.php?title=intel/microarchitectures/coffee_lake&oldid=98251`.

[22] Lucian Cojocar, Jeremie S. Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers". In: *S&P*. 2020.

[23] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor". In: *IEEE Micro* (2010).

[24] Patrick Cronin and Chengmo Yang. "A Fetching Tale: Covert Communication with the Hardware Prefetcher". In: *IEEE HOST*. 2019.

[25] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. "Dont́ Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects". In: *USENIX Security*. 2022.

[26] Guillaume Didier and Clémentine Maurice. "Calibration Done Right: Noiseless Flush+Flush Attacks". In: *DIMVA*. 2021.

[27] Guillaume Didier, Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour. "Characterizing Prefetchers using CacheObserver". In: *SBAC-PAD*. 2022.

[28] Jack Doweck. “Inside Intel Core microarchitecture”. In: *Hot Chips*. 2006.

[29] Jack Doweck. *Inside Intel Core Microarchitecture and Smart Memory Access*. Tech. rep. Intel, 2006.

[30] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. “Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake”. In: *IEEE Micro* (2017).

[31] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *ASPLOS*. 2018.

[32] Babak Falsafi and Thomas F. Wenisch. *A Primer on Hardware Prefetching*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2014.

[33] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. “SQUIP: Exploiting the Scheduler Queue Contention Side Channel”. In: *S&P*. 2023.

[34] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. “AutoLock: Why Cache Attacks on ARM Are Harder Than You Think”. In: *USENIX Security*. 2017.

[35] Daniel Gruss. “Transient-Execution Attacks”. Habilitation thesis. Graz University of Technology, 2020.

[36] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.

[37] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security*. 2015.

[38] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games - Bringing Access-Based Cache Attacks on AES to Practice”. In: *S&P*. 2011.

[39] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. “Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks”. In: *S&P*. 2022.

[40] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. “Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems”. In: *MICRO*. 2009.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. 6th edition. Morgan Kaufmann, 2019. ISBN: 978-0-12-811905-1.

[42] M.D. Hill and A.J. Smith. “Evaluating associativity in CPU caches”. In: *IEEE Transactions on Computers* (1989).

[43] Ralf Hund, Carsten Willems, and Thorsten Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *S&P*. 2013.

[44] Ibrahim Hur and Calvin Lin. “Memory Prefetching Using Adaptive Stream Detection”. In: *MICRO*. 2006.

[45] *Intel ® Pentium ® Processor Invalid Instruction Erratum Overview*. Consolidated in 2015, CVE-1999-1476. Intel Corporation. 1997. URL: https://www.intel.com/content/dam/support/us/en/documents/processors/invalid_instruction_cmpxchg8b_erratum1.pdf.

[46] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 248966-041. Intel Corporation. 2018. URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[47] *Intel 64 and IA-32 Architectures Software Developer's Manual*. 325462-076US. Intel Corporation. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[48] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors". In: *Euromicro DSD*. 2015.

[49] Bruce L. Jacob and Trevor N. Mudge. "Virtual memory in contemporary microprocessors". In: *IEEE Micro* (1998).

[50] Víctor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. "Making data prefetch smarter: adaptive prefetching on POWER7". In: *PACT*. 2012.

[51] Norman P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers". In: *ISCA*. 1990.

[52] Norman P. Jouppi. *WRL-TN-14: Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*. Tech. rep. DEC Western Research Laboratory, 1990. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.4913.

[53] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ISCA*. 2014.

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[55] Francois Koeune and Jean-Jacques Quisquater. *A timing attack against Rijndael*. Tech. rep. 1999.

[56] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. "RAMBleed: Reading Bits in Memory Without Accessing Them". In: *S&P*. 2020.

[57] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. "IBM POWER6 microarchitecture". In: *IBM Journal of Research and Development* (2007).

[58] Ankur Limaye and Tosiron Adegbija. "A Workload Characterization of the SPEC CPU2017 Benchmark Suite". In: *ISPASS*. 2018.

[59] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security*. 2016.

[60] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take A Way: Exploring the Security Implications of AMD’s Cache Way Predictors”. In: *ASIA CCS*. 2020.

[61] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. “PLATYPUS: Software-based Power Side-Channel Attacks on x86”. In: *S&P*. 2021.

[62] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *USENIX Security*. 2018.

[63] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *S&P*. 2015.

[64] Adam Malamy, Rajiv N. Patel, and Norman M. Hayes. “Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature”. US5353425A. 1992.

[65] Hector Martin. *M1ssing Register Access Controls Leak EL0 State*. Viewed on 2022-08-02, CVE-2021-30747. 2021. URL: `https://m1racles.com`.

[66] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “C5: Cross-Cores Cache Covert Channel”. In: *DIMVA*. 2015.

[67] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *RAID*. 2015.

[68] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *NDSS*. 2017.

[69] John D. McCalpin. “Address Hashing in Intel Processors”. In: *IXPUG*. 2018.

[70] John D. McCalpin. *Mapping Core and L3 Slice Numbering to Die Location in Intel Xeon Scalable Processors*. Tech. rep. 2021.

[71] Nimrod Megiddo and Dharmendra S. Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache”. In: *FAST ’03 Conference on File and Storage Technologies*. 2003.

[72] Pierre Michaud. “Best-offset hardware prefetching”. In: *HPCA*. 2016.

[73] Sparsh Mittal. “A Survey of Recent Prefetching Techniques for Processor Caches”. In: *ACM Computing Surveys* (2016).

[74] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. “Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System”. In: *PACT*. 2009.

[75] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. “Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture”. In: *International Conference on Parallel Processing, ICPP*. 2015.

[76] David Monniaux and Valentin Touzeau. “On the Complexity of Cache Analysis for Different Replacement Policies”. In: *J. ACM* (2019).

[77] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* (1965).

[78] Trevor N. Mudge. “Power: A First-Class Architectural Design Constraint”. In: *Computer* (2001).

[79] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 2nd edition. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[80] John von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* (1993).

[81] Hamed Okhravi, Stanley Bak, and Samuel T. King. “Design, implementation and evaluation of covert channel attacks”. In: *IEEE International Conference on Technologies for Homeland Security (HST)*. 2010.

[82] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. “The Case for a Single-Chip Multiprocessor”. In: *SIGPLAN Not.* (1996).

[83] Philipp Oppermann. *Writing an OS in Rust*. Viewed on 2022-09-27. 2018. URL: `https://os.phil-opp.com`.

[84] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *CT-RSA*. 2006.

[85] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. “Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical”. In: *USENIX Security*. 2021.

[86] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. “Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical”. In: *arXiv:2103.03443* (2021). Extended version of the USENIX Security paper.

[87] Samuel Pakalapati and Biswabandan Panda. “Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching”. In: *ISCA*. 2020.

[88] Subbarao Palacharla and Richard E. Kessler. “Evaluating Stream Buffers as a Secondary Cache Replacement”. In: *ISCA*. 1994.

[89] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface*. Risc V edition. 2018. ISBN: 978-0-12-812275-4.

[90] Colin Percival. “Cache missing for fun and profit”. In: *BSDCan*. 2005.

[91] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security*. 2016.

[92] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. “Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks”. In: *CCS*. 2021.

[93] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds”. In: *CCS*. 2009.

[94] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. “Reverse Engineering the Stream Prefetcher for Profit”. In: *EuroS&P Workshops*. 2020.

[95] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. “Port Contention Goes Portable: Port Contention Side Channels in Web Browsers”. In: *ASIA CCS*. 2022.

[96] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. “SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers”. In: *EuroS&P*. 2021.

[97] Satish Kumar Sadasivam, Brian W. Thompto, Ronald N. Kalla, and William J. Starke. “IBM Power9 Processor Architecture”. In: *IEEE Micro* (2017).

[98] Anish Saxena and Biswabandan Panda. “DABANGG: A Case for Noise Resilient Flush-Based Cache Attacks”. In: *S&P Workshops*. 2022.

[99] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. “RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches”. In: *EUROSEC*. 2017.

[100] Mark Seaborn and Thomas Dullien. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. Viewed on 2022-08-02. Google Project Zero, 2015. URL: `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`.

[101] André Seznec. “A new case for the TAGE branch predictor”. In: *MICRO*. 2011.

[102] André Seznec. “Branch Predictors”. In: *Encyclopedia of Parallel Computing*. 2011.

[103] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. “On the effectiveness of address-space randomization”. In: *CCS*. 2004.

[104] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Predictor-directed stream buffers”. In: *MICRO*. 2000.

[105] Young-joo Shin, Hyung Chan Kim, Dokeun Kwon, Ji-Hoon Jeong, and Junbeom Hur. “Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage”. In: *CCS*. 2018.

[106] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 10th edition. Wiley Publishing, 2018. ISBN: 978-1-118-06333-0.

[107] Balaram Sinharoy, Ronald N. Kalla, Joel M. Tendler, Richard J. Eickemeyer, and Jody B. Joyner. “POWER5 system microarchitecture”. In: *IBM Journal of Research and Development* (2005).

[108] Balaram Sinharoy, James Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, José E. Moreira, D. Levitan, S. Tung, David Hrusecky, James W. Bishop, Michael Gschwind, Maarten Boersma, Michael Kroener, Markus Kaltenbach, Tejas Karkhanis, and K. M. Fernsler. “IBM POWER8 processor core microarchitecture”. In: *IBM Journal of Research and Development* (2015).

[109] Alan Jay Smith. "Cache Memories". In: *ACM Computing Surveys* (1982).

[110] Avinash Sodani. "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor". In: *Hot Chips*. 2015.

[111] Avinash Sodani, Roger Gramunt, Jesús Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* (2016).

[112] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers". In: *HPCA*. 2007.

[113] William J. Starke, Jeffrey Stuecheli, David Daly, J. S. Dodson, Florian Auernhammer, Patricia Sagmeister, Guy L. Guthrie, Charles F. Marino, M. S. Siegel, and Bart Blaner. "The cache and memory subsystems of the IBM POWER8 processor". In: *IBM Journal of Research and Devevlopment* (2015).

[114] William J. Starke, Brian W. Thompto, Jeffrey Stuecheli, and José E. Moreira. "IBM's POWER10 Processor". In: *IEEE Micro* (2021).

[115] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. "POWER4 System Microarchitecture". In: *IBM Journal of Research and Development* (2002).

[116] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* (1967).

[117] "Trends and Challenges in the Vulnerability Mitigation Landscape". In: Santa Clara, CA: USENIX Association, 2019.

[118] Vladimir Uzelac and Aleksandar Milenkovic. "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures". In: *ISPASS*. 2009.

[119] Xavier Vera. "Inside Tiger Lake: Intel's Next Generation Mobile Client CPU". In: *Hot Chips*. 2020.

[120] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest". In: *S&P*. 2022.

[121] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data". In: *ISCA*. 2021.

[122] Pepe Vila. "Learning Secrets and Models from Execution Time". PhD thesis. Universidad Politécnica de Madrid (UPM), 2020.

[123] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. "CacheQuery: learning replacement policies from hardware caches". In: *PLDI*. 2020.

[124] Pepe Vila, Boris Köpf, and José F. Morales. "Theory and Practice of Finding Eviction Sets". In: *S&P*. 2019.

[125] Daimeng Wang, Zhiyun Qian, Nael B. Abu-Ghazaleh, and Srikanth V. Krishnamurthy. “PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack”. In: *DAC*. 2019.

[126] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. “Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86”. In: *USENIX Security*. 2022.

[127] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. “A Cache Timing Attack on AES in Virtualization Environments”. In: *Financial Cryptography and Data Security, FC*. 2012.

[128] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th edition. 2010. ISBN: 0321547748.

[129] Johannes Wikner and Kaveh Razavi. “RETBLEED: Arbitrary Speculative Code Execution with Return Instructions”. In: *USENIX Security*. 2022.

[130] Zhenyu Wu, Zhang Xu, and Haining Wang. “Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud”. In: *IEEE/ACM Transactions on Networking* (2015).

[131] William A. Wulf and Sally A. McKee. “Hitting the memory wall: implications of the obvious”. In: *SIGARCH Computer Architecture News* (1995).

[132] Wenjie Xiong and Jakub Szefer. “Survey of Transient Execution Attacks and Their Mitigations”. In: *ACM Computing Surveys* (2021).

[133] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh R. Joshi, Matti A. Hiltunen, and Richard D. Schlichting. “An exploration of L2 cache covert channels in virtualized environments”. In: *Cloud Computing Security Workshop, CCSW*. 2011.

[134] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. “Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World”. In: *S&P*. 2019.

[135] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security*. 2014.

[136] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. “Mapping the Intel Last-Level Cache”. In: *IACR Cryptololgy ePrint Archive* (2015).

[137] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: A Timing Attack on OpenSSL Constant Time RSA”. In: *CHES*. 2016.

[138] Kenneth C. Yeager. “The Mips R10000 superscalar microprocessor”. In: *IEEE Micro* (1996).

## RÉSUMÉ

Les caches contribuent de façon majeure à la performance des processeurs modernes. Pour réduire le nombre d'accès manquant le cache *(cache miss)*, les fabricants incluent des *prefetchers* ou, en français, préchargeurs, qui visent à anticiper les requêtes mémoires du processeur. Toutefois, les fabricants ne documentent guère ces préchargeurs, d'où l'intérêt d'en faire la rétro-ingénierie. Ceci intéresse les chercheurs en sécurité autant que la communauté haute performance.
Néanmoins, cette rétro-ingénierie est rendue difficile parce que les préchargeurs opèrent sur le cache et sont entrainés par les accès mémoires, moyen principal de mesurer l'état du cache. L'observateur interfère alors avec l'expérience.
Pour éviter cela, nous avons cherché à employer un autre canal auxiliaire, Flush+Flush, qui utilise l'instruction `clflush`. Cette instruction n'est pas un accès mémoire, et n'influence donc pas les préchargeurs. Toutefois, une importante variabilité et un fort bruit nuisent grandement à cette primitive sur les processeurs modernes. Notre première contribution a été d'identifier la source du bruit et de la variabilité comme provenant du réseau d'interconnexion entre les cœurs. Nous avons identifié sa topologie et modélisé les temps d'exécution de `clflush`, ce qui nous a permis de concevoir un algorithme de calibration qui rend Flush+Flush aussi précise que Flush+Reload. Cette primitive améliorée a rendu notre stratégie originale viable, et nous avons alors atteint notre deuxième contribution. Nous avons développé un outil, *CacheObserver*, écrit en Rust, qui utilise Flush+Flush pour surveiller l'état du cache dans une région de la mémoire, en réaction a une séquence d'accès mémoires. À l'aide de celui-ci, nous avons mis à jour des comportements jusque-là ignorés du *L2 Stream prefetcher* des processeurs Intel Whiskey et Coffee Lake. Nous avons aussi montré que ce préchargeur interagit avec l'autre préchargeur de niveau 2 de ces processeurs, le *Adjacent Cache Line Prefetcher*, préchargeur de ligne de cache adjacente.

## MOTS CLÉS

Rétro-ingénierie, Micro-architecture, Caches, Prefetchers, Préchargeurs mémoires

## ABSTRACT

Caches are essential to the performance of modern CPUs. *Hardware prefetchers* attempt to fetch lines into the cache before these are requested. This aims at reducing the number of cache misses, especially *cold misses*. Most modern CPU designs include prefetchers, but the manufacturers disclose very little about those prefetchers. However, prefetchers may have a security impact, potentially leaking private information. Consequently, uncovering and documenting their behavior is valuable to the academic communities in security and high-performance computing.
Nevertheless, this endeavor is challenging. Prefetchers only affect the cache state, usually deduced by timing memory accesses. Unfortunately, memory accesses influence the prefetchers; hence this method interferes with the experiments. To work around this issue, we started with the idea of using the Flush+Flush cache channel, which uses the `clflush` instruction. This instruction is not a memory access; hence, it does not influence the prefetchers. However, significant variability and noise hamper this primitive on modern CPUs. Our first contribution was to identify the interconnect between cores as the source of this noise, model `clflush` execution time, and use this knowledge to improve Flush+Flush to be as accurate a primitive as the reliable Flush+Reload. This made our initial strategy viable to achieve our second contribution. We built a framework, *CacheObserver*, written in Rust, that uses Flush+Flush to monitor the cache state of a range of addresses in response to a sequence of memory accesses. Using this framework, we uncovered behavior of the L2 Stream prefetcher on Intel's Coffee Lake and Whiskey Lake CPUs. We also showed that this prefetcher interacted with the L2 adjacent cache line prefetcher, the other L2 prefetcher included in those CPUs.

## KEYWORDS

Reverse engineering, Microarchitecture, Caches, Prefetchers