Characterizing Prefetchers using CacheObserver

Guillaume Didier
DGA; Univ. Rennes, CNRS, IRISA;
DIENS, École normale supérieure, PSL
Rennes & Paris, France
guillaume.didier@inria.fr

Clémentine Maurice
Univ. Lille, CNRS, Inria
Lille, France
clementine.maurice@inria.fr

Antoine Geimer
Univ. Lille, CNRS, Inria
Lille, France
antoine.geimer@inria.fr

Walid J. Ghandour
Univ. Lille, CNRS, Inria
Lille, France
ghandour@utexas.edu

***Abstract*—Hardware prefetchers are mostly undocumented micro-architectural components of the cache hierarchy, with performance and security implications. Intel CPUs feature four named prefetchers whose behaviors are generally unknown. We build CacheObserver, a generic framework to study prefetchers on Intel CPUs and the first to use the Flush+Flush primitive, unlike previous studies using Flush+Reload. We apply this framework to characterize the L2 Stream prefetcher on the Intel Whiskey and Coffee Lake micro-architectures and uncover the behavior of the L2 Stream prefetcher upon the first few accesses in a page. We also uncover interactions at the L2 level between the Stream and the Adjacent Cache Line prefetchers.**

***Index Terms*—Hardware prefetchers, reverse-engineering, clflush, L2 cache, rust, Stream prefetcher, Flush+Flush.**

I. Introduction

Caches are a key component of micro-architectures used to overcome the “memory wall” between the speed of CPUs and DRAM main memories. To improve the performance by reducing the number of *cold* misses, manufacturers include prefetchers in their caches. These components predict future memory accesses and fetch the corresponding data in the cache ahead of the program requests. These prefetchers give CPUs manufacturers competitive advantages, hence the sparse disclosure of their existence and undocumented behavior.

Caches have significant performance implications but sit below the divide between architecture and micro-architecture. Hence, Instruction Set Architectures (ISAs) include no way to determine the caches' content. Thus this state must be measured through timing information. Previous studies of the various prefetchers have been non-systematic, timing memory accesses to infer the cache state [2], [25], [27]. Yet understanding prefetcher is helpful to improve program performance as well as to assess micro-architecture security.

However, memory accesses may trigger the prefetcher, and thus observations may affect the experimental results. Thus, we set out to build a framework enabling a more systematic study of such components, using the Flush+Flush primitive to infer the cache state with minimal interference and apply it to Intel CPUs. We focus on L2 prefetchers, which are more complex and less studied than the L1 ones.

This research attempts to answer the following questions:

- (**Q1**) How can we build a detailed view of prefetch activity?
- (**Q2**) What is the impact on prefetch activity of using load instructions for measurements compared with `clflush`?
- (**Q3**) How does the Intel L2 Stream prefetcher behave, especially on the first few accesses in a page?
- (**Q4**) 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 with each other.

Outline: Section II presents background information and related work. Section III then describes the CacheObserver framework and Section IV the setup of the experiments we ran. Section V presents the experimental results on the L2 Stream prefetcher and Section VI those on the other L2 prefetcher and prefetcher interactions. Section VII discusses the advantages and limitations of our approach while Section VIII sums up the results and future line of investigations.

II. Background and Related Work

A. Cache hierarchy on modern CPUs

The *memory wall* refers to the increasing speed difference between CPUs and DRAM main memory. DRAM access times in CPU cycles have increased by orders of magnitude in the last few decades, reaching hundreds of cycles. To compensate this slowdown, *caches* are used, leveraging the “make the common case fast” principle [10], [11]. Organized as a hierarchy of increasingly smaller but faster SRAM memories, they retain values that have been recently used. They are not architecturally visible and are part of the micro-architecture.

The CPUs we studied belong to a lineage that starts with the Sandy Bridge CPUs (2010) and whose latest descendants are the Comet Lake CPUs (2020) [36]. These CPUs have a 3-level cache hierarchy. Each core possesses a private first-level instruction (L1$) and data (L1D) cache and a second-level private cache (L2), queried after either L1. A shared third-level cache (L3), structured in slices, then serves as the last-level cache [3]. The L3 is *inclusive* of the lower levels of caches in all CPUs in this lineage from 2010 to 2020. However, newer micro-architectures deviate from this set-up and introduce non-inclusive caches in the Skylake-SP server CPUs and Ice Lake and Rocket Lake client CPUs [13], [14].

TABLE I: Prefetchers disclosed by Intel for Sandy Bridge CPUs, in [14] Vol. 4 p 2-179.

Bit	Intel Description
0	L2 Hardware Prefetcher Disable (R/W) 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) 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) 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) 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.

A *cache hit* occurs when a program requires data present in a cache, such a request is fulfilled significantly faster than requests for data only found in slower levels of the memory hierarchy, *i.e.*, slower caches or main memory. The opposite is called a *miss*. Misses can be classified according to their cause. *Cold misses* occur on the first access to a piece of data. Other misses (conflict, coherence, capacity) are caused by the eviction of data that was previously cached [10], [11].

Those caches manipulate data grouped in contiguous blocks of data called *cache lines*, usually 64 bytes on the x86 architecture [14] (implemented by the Intel CPUs we studied). The size of the block helps leverage some amount of *spatial locality*: "programs tend to use data close to data they used recently" [10]. For instance, let's consider a program sequentially accessing a large buffer of bytes. In this case, the 63 neighboring bytes of a recently used byte are part of the same line. They are brought into the cache with the first requested byte, and requests to these bytes then hit the cache.

However, the above example still results in one miss per 64 bytes accessed. Yet, this pattern appears to be quite predictable; hence the introduction of a mechanism to predict future accesses or misses and request the data from the main memory before the program actually requests them.

B. Prefetchers on modern Intel CPUs

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. Table I describes them. Additionally, Appendix E, Section 2.5.4 in [13] gives more details about Sandy Bridge prefetchers, notably calling the L2 Hardware Prefetcher a stream prefetcher.

Jouppi [16] introduced *stream prefetchers*: A *streams* is a sequence of accesses to consecutive lines in increasing or decreasing order. He later defined prefetching of *quasi streaming patterns* where some lines are skipped [17]. Academic papers published variations of the concept [12], [23], [26], [29]. Some industry disclosures hint at their implementations [4]–[6], [19], [28], [30]. Two parameters characterize prefetchers: *distance*, *i.e.*, how far from the current line do they fetch, and *degree*, *i.e.*, how many lines do they fetch upon a single access [25].

Rohan et al. [25] were the first to study the L2 Stream prefetcher. Their reverse-engineering on Kaby Lake CPUs confirmed many of the prefetcher's properties Intel disclosed [13]. Precisely, the prefetcher fetches consecutive cache lines in the positive (increasing addresses) or negative (decreasing addresses) direction. One may start a stream and trigger prefetches with an access further away in the same page. They also estimated the prefetcher tracks up to 16 pages, consistently with the Intel statement that both a positive and a negative stream can be tracked per page for a total of 32 streams. They built a covert channel exploiting the prefetcher state sharing by hyper-threads. Lastly, they showed it does not cross $4\,\mathrm{KiB}$ page limits, even with $2\,\mathrm{MiB}$ huge pages.

Additionally, two studies [2], [27] of the L1 IP-based Stride prefetcher unveiled security implications of this prefetcher. It reacts to victim memory accesses and can insert control-flow-dependent cache hits, enabling side-channel attacks [27]. Its state sharing among hyper-threads also enables a channel [2].

C. `clflush` and its use as a cache side-channel primitive

The unprivileged x86 `clflush` instruction [14] ensures the latest value of a cache line is written back to memory and evicted from the whole cache coherence domain. On modern x86 CPUs, this includes all caches on all the sockets in the system. It also helps build cache channels on this architecture.

1) Flush+Reload: Load execution time depends on where a requested line is found in the memory hierarchy and, thus, leaks whether a line is cached. Using the `clflush` instruction to put lines in a known state, the Flush+Reload primitive then measures load timing [37]. Although both fast and low-noise, its loads may interfere with the prefetcher and increase the number of prefetches. For our study, we thus sought a different measurement primitive that wouldn't trigger prefetches.

2) Flush+Flush: On Intel CPUs, previous work [3], [9] showed that `clflush` execution time depends on the cache coherence state of the line. Thus, it can check if a line is cached anywhere in the cache coherence domain with no memory access. We re-use the calibration method in [3], increasing the primitive's reliability.

D. Reverse engineering of other components

1) Memory Hierarchy: Reverse engineering parts of the Intel cache hierarchy is the foundation upon which reliable side channels are built [3], [21], [22], [34]. In particular, several works uncovered the addressing functions of Intel CPUs' last-level caches [15], [21], [38]. Pessl et al. [24] devised two different methods to reverse engineer DRAM addressing schemes and used these to build cross socket channels requiring no shared memory. Their insights also improved Rowhammer attacks on DDR4. Lipp et al. [20] uncovered the L1D μtag hash function and cache-way predictor in AMD CPUs from 2011 to 2019 and constructed new cache channels using these structures between sibling hyper-threads.

Of notable interest is the framework built by Vila et al. to reverse engineer cache replacement policies in various CPUs. This work first required them to find reliable methods to compute eviction sets [35], which were then used by the main framework [34]. Eviction sets are sets of addresses that, accessed in a sequence, cause the eviction of a specific cache line. Green et al. [8] uncovered an undocumented policy in various ARM CPUs related to cache eviction that hampers eviction-based cache channels, and Van Schaik et al. [32] reverse-engineered page-table caches, used to speed up page-table walks on CPUs from Intel, AMD and ARM.

Additionally, Paccagnella et al. [22] uncovered low-level details of the ring interconnect of Intel CPUs of the Sandy bridge lineage and used this to construct a contention side channel, unlike most micro-architectural channels relying on persistent state. Recently, Vicarte et al. [33] have uncovered a data-dependent prefetcher on Apple's M1 and A14 CPUs and shown it can be exploited to break kernel ASLR.

2) Branch predictors: Spectre v1 [18] relies on knowledge of the branch predictor, another micro-architectural component that has been studied. So does the BranchScope [7] attack, using the branch direction predictor as the micro-architectural state that leaks information to the attacking thread.

Uzelac et al. [31] proposed a methodology to reverse-engineer branch predictors. Bhattacharya et al. [1] reconstructed the branch direction predictors of Intel CPUs from Nehalem to Broadwell and showed their consistency with 2-bit (Nehalem) or 3-bit saturating counters. This insight led to a side-channel attack on blinded scalar multiplication.

III. 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 micro-architectural and cannot be queried directly; it is usually measured by timing memory loads. However, such loads may trigger prefetches and interfere with the experiments [2], [25], [27]. Hence our choice of the Flush+Flush primitive, which times the `clflush` instruction.

Our CacheObserver[1] framework re-uses our previous code base[2] [3], implementing reliable calibration of Flush+Flush channels, both are written in Rust. The core of the framework is the *Prober* object. It 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).

A. Prober

Using load instructions, we can only probe one line per experiment run and must then repeat it 64×, once per offset in the page. If we assume `clflush` does not alter the prefetchers state, a single run of a pattern can give the state of all 64 cache lines. We thus implement three strategies:

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

[1]https://github.com/MIAOUS-group/CacheObserver

[2]https://github.com/MIAOUS-group/calibration-done-right

```
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 1: Measurement primitives.

2) `SingleFlush`: This strategy probes a single address with `clflush` after each run of the pattern. 64 iterations are required to cover the full page.
3) `SingleReload`: This baseline strategy is similar to `SingleFlush` but probes with load instructions. It is the only strategy usable with loads, as more measurement loads would interfere with the prefetcher.

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

We warm up over 100 iterations and collect data over 1024 iterations. According to Rohan et al. [25] and the Intel manuals [13], [14], cycling through a sufficient number of pages should evict any lingering entry and reset the prefetcher state for the page. We thus select a fresh page from a shuffled queue of 63 and fully flush it before each pattern run to reset the prefetcher.

Modern CPU's out-of-order execution means extra serialization is needed to ensure the correct measurement of execution times. Listing 1 presents the measurement code. **`fn`** `only_flush` is used for `SingleFlush` and `FullFlush`, while **`fn`** `only_reload` is used for `SingleReload`.

B. Access patterns

We study L2 caches that deal with memory accesses at cache line granularity. We thus identify accesses as cache line offsets within a page, from 0 to 63. An access pattern is a list of such offsets combined with the function used for each access. This feature is meant to study prefetchers that take into account the memory accesses' Instruction Pointers (IP).

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).

All experiments other than 0 and 1 result in cubes (i, j, p) giving for each pattern $f(i, j)$ the hit rate of each line (probe p). Cubes cannot be printed, so we derive two kinds of figures:

Total prefetch in page (e.g., fig. 1a): 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. 1b): 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. Further patterns could be used to study other prefetchers or behaviors. Figures 1, 3 and 4 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 line of hits. 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 clarify this, we go through a pair of slice figures, with no prefetchers enabled: E_4 and D_4:

– E_4 executes the pattern $(i, i+j, i+2j, i+3j, i+4j)$. Without 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 in the sequence access the same line). Each column of the figure contains up to 5 points. On the slice with $i = 15$, fig. 1b 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. 1c, 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 **Q1**: Prefetched lines are any high hit rate lines that are not in the pattern motif.

IV. Experimental setup

A. Hardware and software configuration

We use two machines with different micro-architectures:

WKL a Dell Latitude 7400, Intel Core i5-8365U (Whiskey Lake micro-architecture, 4 physical cores, 8 hyper-threads), running Fedora 30.

CFL a Dell Precision 3630, Intel Core i9-9900 (Coffee Lake, 8 cores, 16 threads), running Ubuntu 18.04.5 LTS.

Both CPUs are minor refreshes of the Kaby Lake architecture; however, our results suggest their prefetchers differ slightly.

To stabilize the frequency of the CPUs, we disable turbo-boost and set the `cpupower` governor to `performance`.

B. Control group

To ensure the prefetcher is the cause of our observations, we run identical experiments with all prefetchers disabled by MSR 420. As figs. 1a to 1c show, 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.

V. The L2 Stream prefetcher

A. Proposed data structure of the prefetcher

We assume this L2 prefetcher ignores offset bits used to select bytes inside lines and uses physical addresses. We thus identify cache lines in a page with numbers in $[\![0, 63]\!]$.

Our results lead us to propose the structure below, illustrated in fig. 2, which seems consistent with our observations and to explain behaviors observed by [25], even though incomplete:

– A table of *stream entries*, as suggested by [25], 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 elucidated the structure of these states. The *last fetched line* (L) is the last prefetch candidate or the last request when lacking such a candidate.

– The *prefetch candidate* (❶) logic block takes as an input the miss from the L1 cache and the stream entry for the same page. It uses the stream direction state and *last fetched line* to suggest prefetch candidates and update the stream entry.

– The *prefetch confidence* (❷) logic block computes and updates the confidence prefetcher for this stream based on requests made and whether they fit the stream.

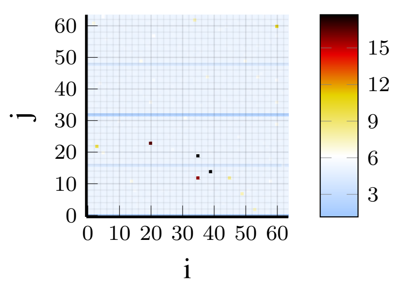
– The *prefetch arbitration* (❸) logic uses the confidence and candidates from all prefetchers to pick which to issue.

B. Experimental results

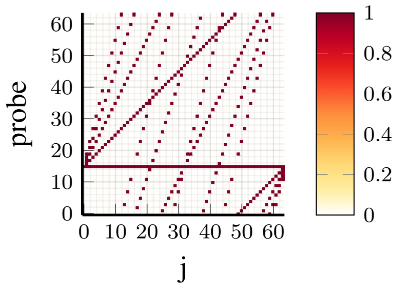
While the Whiskey and Coffee Lake CPUs differ in detail, the same general principles apply. However, the prefetch arbitration in Coffee Lake seems to prefetches more aggressively than the Whiskey Lake one. This prefetcher fetches from main memory into L2 and, by inclusivity, L3, or sometimes only L3. In addition, Whiskey Lake has a smaller L3 cache. Thus, the L3 size cannot be excluded as a reason for such differences.

1) First access behavior: Prefetch may occur if the first access in a page is in $\{0, 1, 62, 63\}$, *i.e.*, the page's first or last two lines, (figs. 1d and 1e). Lines up to line 6 (resp. down to 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 to line 2 or 61. On Coffee Lake, this prefetch always occurs (fig. 1e).

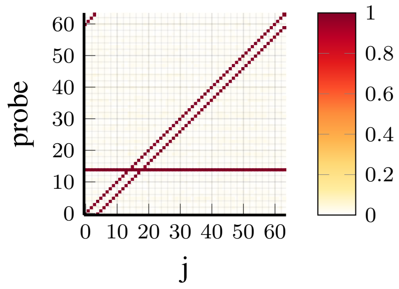
2) Subsequent accesses prefetch 0 or 2 lines: Subsequent accesses prefetch 0 or 2 consecutive lines adjacent to the *last fetched line* or the current access, as seen in figs. 1, 3 and 4 when only the Stream prefetcher is enabled. Section 3.3 of [25] claims the prefetch distance is 1 to 4, and the degree is



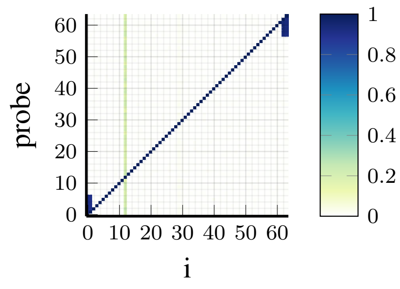
a: WKL, exp. E_4, $(i, i+j, \ldots, i+4j)$ with no prefetcher: average number of hit in the page for i and j.



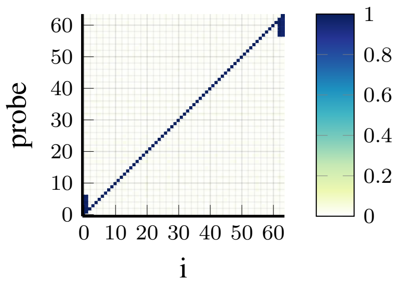
b: WKL, exp. E_4, $(i, i+j, \ldots, i+4j)$ with no prefetcher: hit rate in each line in the page for j, with $i = 15$.



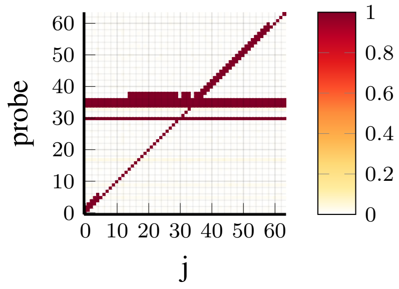
c: CFL, exp. D_4, $(i, j, j-4)$ with no prefetcher: hit rate in each line in the page for j, with $i = 14$.



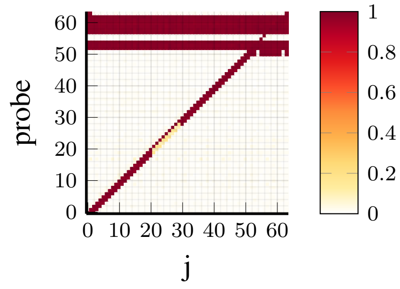
d: WKL, exp. 1, (i) with Stream prefetcher: hit rate in each line in the page for i.



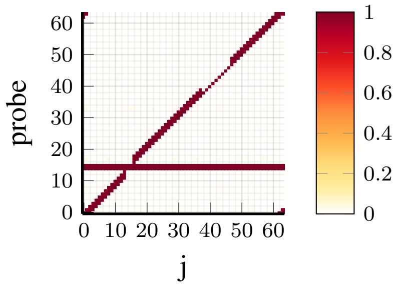
e: CFL, exp. 1, (i) with Stream prefetcher: hit rate in each line in the page for i.



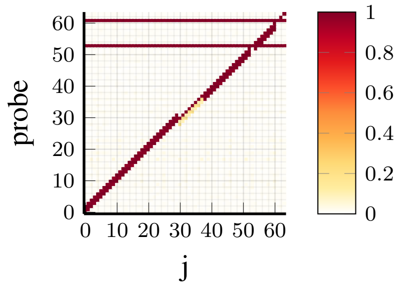
f: WKL, exp. A_4, $(i, i+4, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 30$.



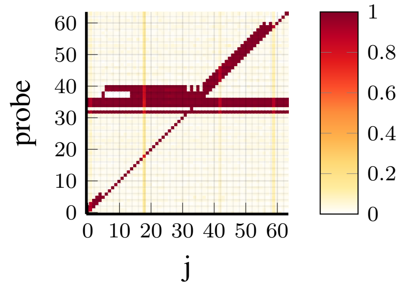
g: CFL, exp. B_8, $(i, i-8, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 62$.



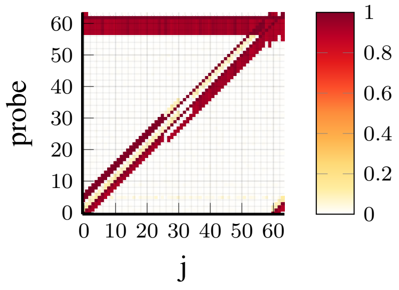
h: WKL, exp. B_1, $(i, i-1, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 15$.



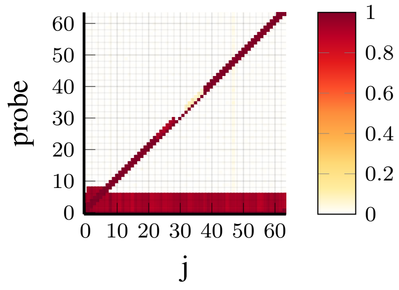
i: CFL, exp. B_8, $(i, i-8, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 61$.



j: WKL, exp. F_2, $(i, i+2, j, i+4)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 32$.

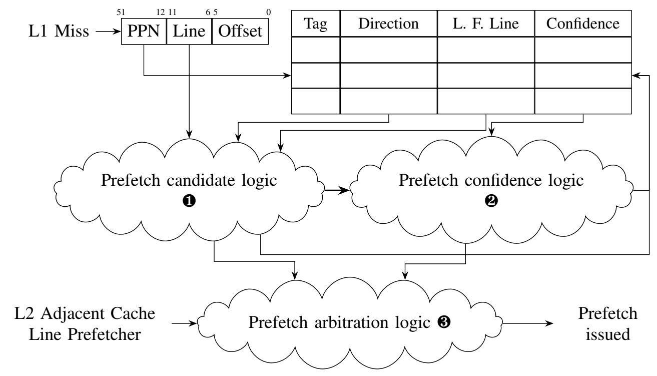


k: WKL, exp. C_4, $(i, j, j+4)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 62$.



l: WKL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 0$.

Fig. 1: Experimental results, see section III-B for 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`.



L1 Miss
51
12 11
6 5
0
PPN
Line
Offset
Tag
Direction
L. F. Line
Confidence
Prefetch candidate logic ❶
Prefetch confidence logic ❷
L2 Adjacent Cache Line Prefetcher
Prefetch arbitration logic ❸
Prefetch issued

Fig. 2: The proposed structure for the Stream prefetcher.

4 to 8. However, we show several prefetches 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 next 3 accesses. This insight comes from detecting the precise effect of patterns on the cache and that of their prefixes to determine what access causes which prefetch.

3) Location of prefetches: Prefetches occur in either a positive or negative direction as seen in figs. 1f and 1g. Positive ones fetch $max(c, L) + 1$, $max(c, L) + 2$ with c the *current* line accessed and L the *last fetched line*. Negative ones instead fetch $min(c, L) - 1$ and $min(c, L) - 2$. Using min/max helps when the out-of-order execution shuffles the requests.

4) Stream window and stream reset: This arithmetic is done modulus 64, and thus prefetch wraps around pages near page boundaries, as seen in fig. 1h ($B_1, i = 15$), for $j \in \{0, 1, 62, 63\}$. 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 not deemed by the prefetcher to be part of the stream and seem to reset its stream entry. The direction is then updated towards the positive direction, and two prefetches are issued from the current address with the updated direction, as seen in figs. 1g, 1i and 1j. For a first access in line 62 or 63, the direction is a barely negative (fig. 1k shows an extra access is needed for 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 only accesses in a 62-line *window* around the *last fetched line* are deemed to belong to a stream. [29] includes such a window.

5) Prefetch gap: With a `clflush`-based method, as shown in figs. 1l and 3a to 3f, 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 Coffee and Whiskey Lake regarding the Stream prefetcher.

On Whiskey Lake, fig. 1l ($i = 0$) 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. 1f, 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. 3g and 3h. This shows that the prefetch behavior adapts to the prefetcher's success rate.

6) Prefetch direction: We have not fully elucidated the prefetch direction logic, but we observed the following: The behavior on a first access in lines 0, 1, 62, or 63 are special cases and set the stream direction directly to positive $(0, 1)$ or negative $(62, 63)$, as seen in figs. 3i and 3j. 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. 3k and 3l.

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. 4a and 4b.

7) Prefetch may occur on L2 hits: Looking at fig. 1j ($F2$, $i = 32$), 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 best interpreted comparing it with fig. 3k whose pattern is the underlined prefix of F_2: $(\underline{i, i+2, j}, i+4)$.

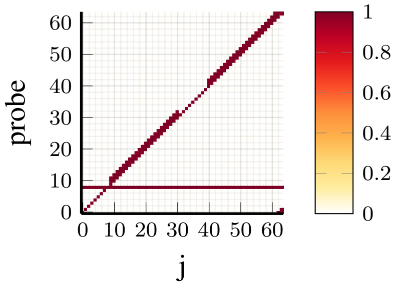
It is worth noting that such an access must still be part of the aforementioned prefetch window, as shown by fig. 4c, where $j \in [\![40, 61]\!]$ do not exhibit prefetches on the fourth access.

8) Suppressed prefetches may still update the prefetcher state: In fig. 4d, when $j \in [\![37, 45]\!]$, we observe prefetches 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 (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, would increase the confidence, and thus triggers prefetches from the *last fetched line* $(j + 2)$, even though a lack of confidence suppressed the $j + 1$ and $j + 2$ prefetches. This is why we proposed a split of the prefetch arbitration logic (❸) in our model.

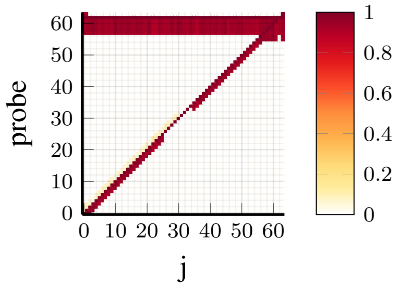
9) Summary: While some uncertainties remain, we get a far clearer picture of this prefetcher's behavior, and our framework can provide further insights using new patterns.

The Stream prefetcher treats in a special 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. Prefetches issued will safely wrap around page limits, which may issue pointless prefetches but causes no potentially dangerous prefetches across page limits.

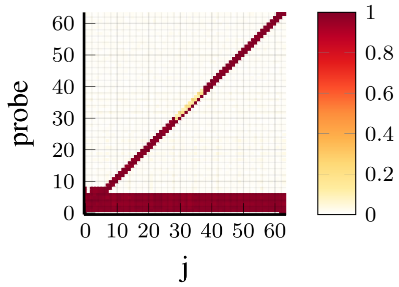
The prefetcher seems to output a confidence metric used to decide whether to prefetch, but suppressed prefetches may update the prefetcher state. Lastly, the prefetcher is reluctant



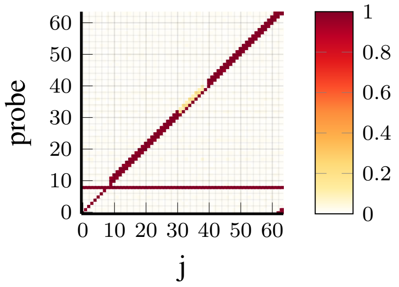
a: WKL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 8$.



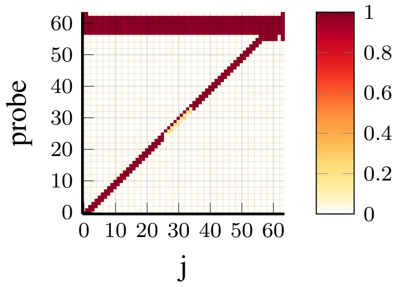
b: WKL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 62$.



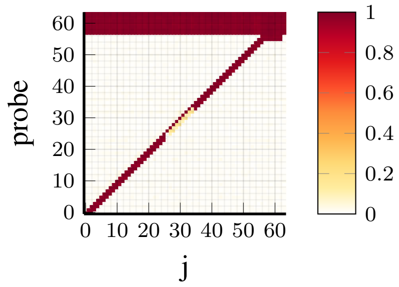
c: CFL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 1$.



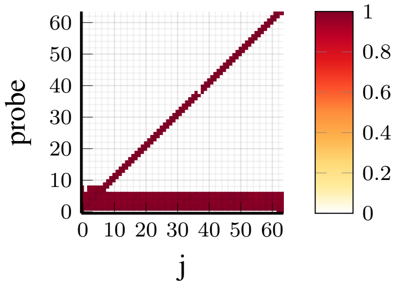
d: CFL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 8$.



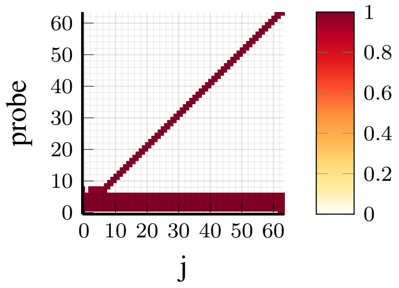
e: CFL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 62$.



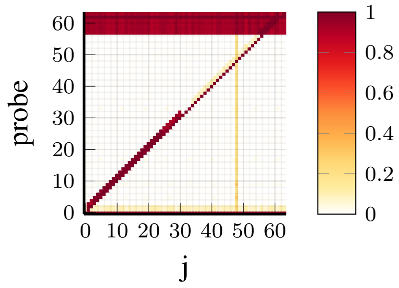
f: CFL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 63$.



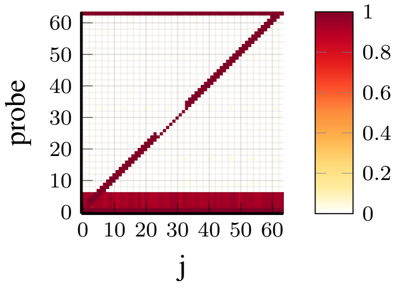
g: `SingleReload` WKL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 1$.



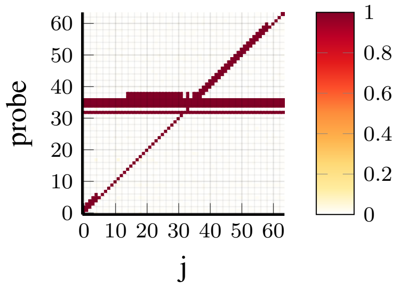
h: `SingleReload` CFL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 1$.



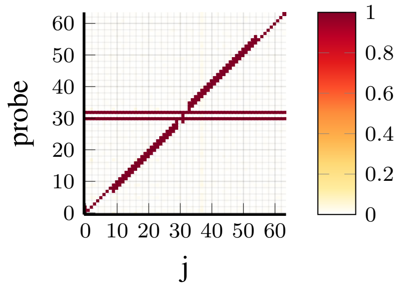
i: WKL, exp. A_2, $(i, i+2, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 62$.



j: WKL, exp. B_2, $(i, i-2, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 1$.

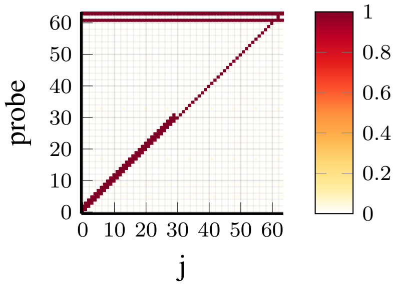


k: WKL, exp. A_2, $(i, i+2, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 32$.

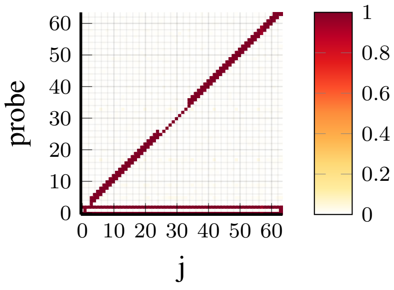


l: WKL, exp. B_2, $(i, i-2, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 32$.

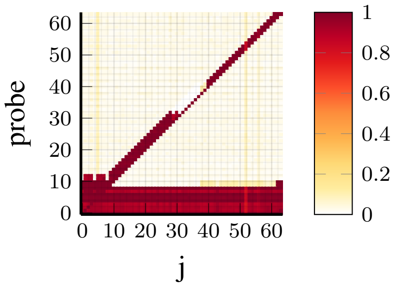
Fig. 3: Experimental results continued from fig. 1. See section III-B for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.



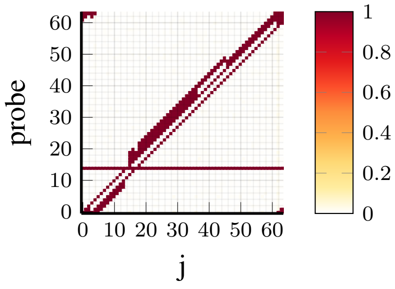
a: WKL, exp. A_2, $(i, i+2, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 61$.



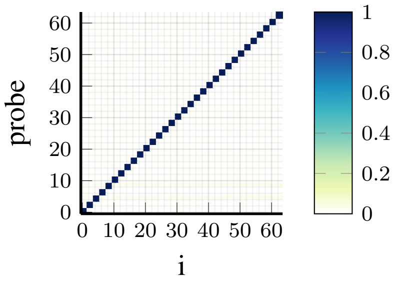
b: WKL, exp. B_2, $(i, i-2, j)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 2$.



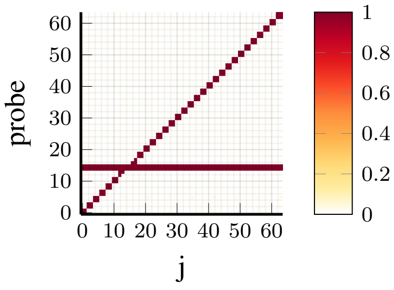
c: WKL, exp. F_2, $(i, i+2, j, i+4)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 0$.



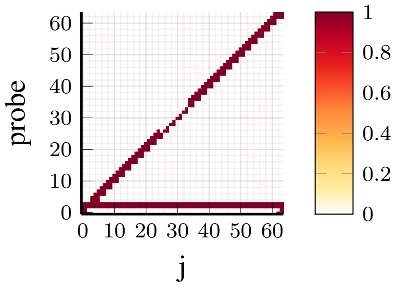
d: WKL, exp. D_3, $(i, j, j-3)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 14$.



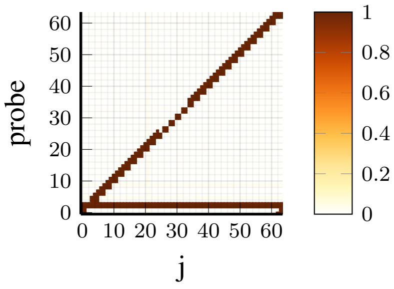
e: WKL, exp. 1, (i), with adjacent line prefetcher: hit rate in each line in the page for i.



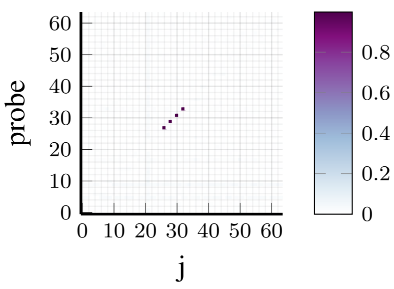
f: WKL, exp. B_1, $(i, i-1, j)$, with adjacent line prefetcher: hit rate in each line in the page for j, with $i = 15$.



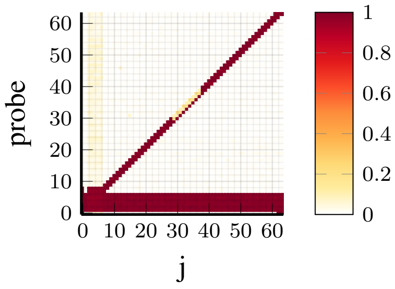
g: WKL, exp. 2, (i, j), with both prefetcher: hit rate in each line in the page for j, with $i = 2$.



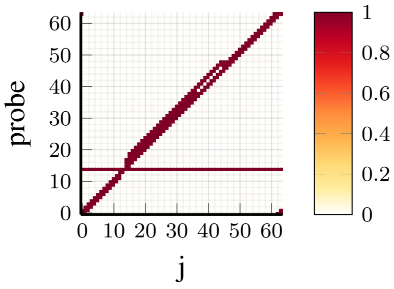
h: WKL, exp. 2, (i, j), maximum of the results for each prefetcher: max hit rate in each line in the page for j, with $i = 2$.



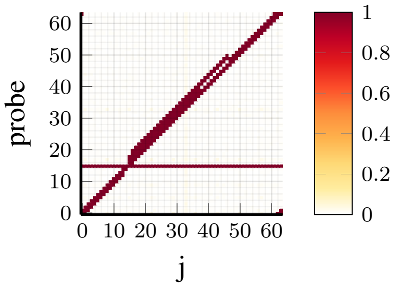
i: WKL, exp. 2, (i, j), difference between the two previous figures in hit rate in each line in the page for j, with $i = 2$.



j: `SingleFlush` CFL, exp. 2, (i, j), with Stream prefetcher: hit rate in each line in the page for j, with $i = 1$.



k: WKL, exp. C_1, $(i, j, j+1)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 14$.



l: WKL, exp. C_1, $(i, j, j+1)$, with Stream prefetcher: hit rate in each line in the page for j, with $i = 15$.

Fig. 4: Experimental results continued from fig. 3. See section III-B for description of the various experiments. Unless specified otherwise, the method used is `FullFlush`.

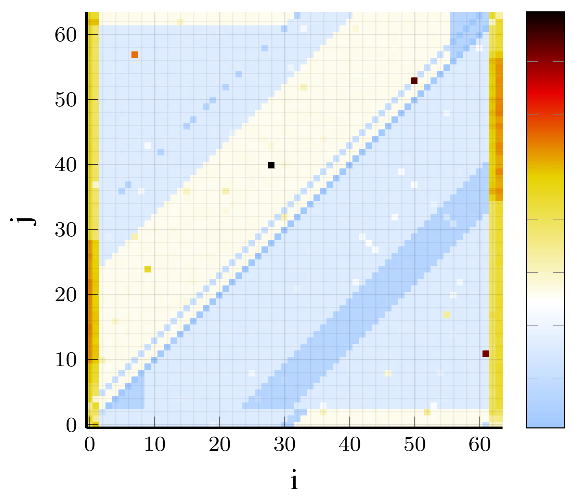


Fig. 5: WKL, exp. D_3, $(i, j, j-3)$ with Stream prefetcher: average number of hit in the page for i and j.

to start streams too close to the page end. These results significantly add to the previous work about **Q3**.

VI. 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. 4e) matches this description. However small differences appear with more accesses, e.g., in fig. 4f, for $j = 13$ or 16, the expected prefetch is missing. Compared with the same figure (1h) 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. 4g, 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. 4h. When comparing the superposition with the experiment with both prefetchers, differences appear, shown in fig. 4i. 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 **Q4** and shows that studying prefetchers in isolation is not enough.

VII. Discussion

A. *Advantages of using* `clflush`

`clflush` evicts lines without using them, whereas a load vindicates the prefetcher in its guess. The latter may increase the prefetcher confidence and/or the arbitration logic trust in the prefetcher, akin to tournament branch prefetchers [10], thereby interfering with measurements.

Our experiments show clear differences between measurements with `SingleReload` (e.g., fig. 3h) and those with `clflush` (figs. 3c and 4j). 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 the `FullFlush` (fig. 3c) and `SingleFlush` (fig. 4j) 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 answer **Q2** showing that loads interfere with measures significantly more than `clflush`. Flush+Flush is both a faster and a more subtle tool for prefetcher reverse-engineering than Flush+Reload and provides more insights.

Lastly, regarding general applicability, Flush+Reload works on all x86 CPUs with an inclusive last-level cache, whereas Flush+Flush is only known to work on Intel CPUs; its status on AMD is unknown. However, Flush+Flush works on all Intel CPUs, including those with non-inclusive caches, thanks to targetting the coherence protocol.

B. *Areas of uncertainty*

Our understanding of Intel L2 prefetchers remains partial.

First, the behavior when an access occurs at $L + 31$ appears unpredictable. Figure 5 shows that along the diagonal $j = i + 31$ the number of prefetches observed is inconsistent. Furthermore, even with an equal number of prefetches, they sometimes occur in different places, e.g., in figs. 4k and 4l (resp. $i = 14$ and 15), 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 $\{0, 1, 62, 63\}$. 90% of the time, we see prefetches of lines up to 6 or down to 57, and the remainder 10% of the time, a behavior consistent with lines 2 and 61, but we ignore 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 an extra access is necessary to start negative streams 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 between the various prefetchers, requires simultaneous investigation of both prefetchers.

C. *Limitations*

CacheObserver is not able to observe L1 prefetches. We initially thought that the L1 prefetchers might not fetch from main memory but only from L2 or L3, with a potential solution being the calibration reloads to determine in which level a line is cached. However, we recently found out that our use of fences prevents activation of the L1 prefetcher [13].

Our framework has information about which access causes which prefetch but has no temporal information on when prefetches are issued and handled. For instance, this precludes understanding how exactly 2 or more requests are emitted per L2 access. For example, those may be emitted in consecutive cycles, but this is not necessarily the design used.

VIII. Conclusion and future work

We present CacheObserver, leveraging 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 prefetch activity. In addition, our technique requires fewer repetitions, 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. This framework could also be used to investigate the memory system reactions to workload memory traces, which could be valuable to study performance issues, in particular prefetcher-induced ones.

Further work: A first 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 direction could be to study the state machine for stream directions to determine its precise structure and the structures used for prefetcher confidence (❷) and arbitration (❸). Lastly, it is unknown whether AMD CPUs are affected by Flush+Flush, and whether the ARM v8 `DC` instructions are constant-time and left available to user-mode by operating systems. Depending on this, our method could be generalized to those architectures.

Acknowledgment

This work has been partly funded by the French Direction Générale de l'Armement, and by the ANR-19-CE39-0007 MIAOUS.

References

[1] S. Bhattacharya, C. Maurice, S. Bhasin, and D. Mukhopadhyay, "Branch prediction attack on blinded scalar multiplication," *IEEE Trans. Computers*, 2020.

[2] Y. Chen, L. Pei, and T. E. Carlson, "Leaking control flow information via the hardware prefetcher," *arXiv:2109.00474*, 2021.

[3] G. Didier and C. Maurice, "Calibration done right: Noiseless flush+flush attacks," in *DIMVA*, 2021.

[4] J. Doweck, "Inside intel core microarchitecture," in *IEEE Hot Chips 18 Symposium (HCS)*, 2006.

[5] ——, "Inside Intel Core microarchitecture and smart memory access," Intel, Tech. Rep., 2006.

[6] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *Micro 50*, 2017.

[7] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS*, 2018.

[8] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "AutoLock: Why cache attacks on ARM are harder than you think," in *USENIX Security*, 2017.

[9] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *DIMVA*, 2016.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 6th Edition*. Morgan Kaufmann, 2019.

[11] M. Hill and A. Smith, "Evaluating associativity in cpu caches," *IEEE Transactions on Computers*, 1989.

[12] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *MICRO-39*, 2006.

[13] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[14] *Intel 64 and IA-32 Architectures Software Developers Manual*, Intel Corporation, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in intel processors," in *Euromicro Conference on Digital System Design (DSD)*, 2015.

[16] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA 17*, 1990.

[17] ——, "WRL-TN-14: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," DEC Western Research Laboratory, Tech. Rep., 1990. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.4913

[18] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

[19] 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," *IBM Journal of R. and D.*, 2007.

[20] M. Lipp, V. Hadzic, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take A way: Exploring the security implications of amd's cache way predictors," in *ASIA CCS*, 2020.

[21] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *RAID*, 2015.

[22] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *30th USENIX Security*, 2021.

[23] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ISCA 21*, 1994.

[24] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX Security*, 2016.

[25] A. Rohan, B. Panda, and P. Agarwal, "Reverse engineering the stream prefetcher for profit," in *EuroS&P Workshops*, 2020.

[26] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *MICRO 33*, 2000.

[27] Y. Shin, H. C. Kim, D. Kwon, J. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *CCS*, 2018.

[28] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Micro 49*, 2016.

[29] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA-13*, 2007.

[30] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of R. and D.*, 2002.

[31] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *ISPASS*, 2009.

[32] S. van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, "Revanc: A framework for reverse engineering hardware page table caches," in *EUROSEC 2017*, 2017.

[33] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *S&P*, 2022.

[34] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, "Cachequery: learning replacement policies from hardware caches," in *PLDI*, 2020.

[35] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *S&P*, 2019.

[36] "Comet Lake - Microarchitectures - Intel - WikiChip," WikiChip LLC, 2020, last edited 2020-07-22. [Online]. Available: https://en.wikichip.org/w/index.php?title=intel/microarchitectures/comet_lake&oldid=97611

[37] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[38] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the intel last-level cache," *IACR Cryptol. ePrint Arch.*, 2015.