Do not Forget Hardware Prefetching When Designing Secure Cache Systems

Bi-annual seminar Report (MS by research)
by

Sumon Nath
(21Q050007)

Supervisor:
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

10 December 2023

Table of Contents

List of Figures

Introduction

Pioneered by Spectre [28] and swiftly followed by other exploits [15; 17; 9; 32; 18], these attacks take advantage of the cache state affected by *transient* instructions. These instructions are speculative, meaning they are not finalized or committed. Given that speculative execution is a fundamental technique utilized by high-performance processors, it cannot be disabled in the pursuit of security.

To counter speculative execution attacks targeting the cache, several proposals [46; 11; 27; 10; 36; 37; 35; 44] aim to enhance security with minimal impact on performance. The essence of these proposals lies in ensuring that speculative instructions either do not affect the cache state or, if they do, make it invisible to the programmer (referred to as invisible speculation). Among these, GhostMinion [11] stands out as the most stringent, addressing not only typical attacks but also backward-in-time threats like speculative interference attacks [15].

GhostMinion enforces a strict ordering mechanism to thwart various speculative execution attacks throughout the cache system, encompassing the cache hierarchy, miss status holding registers (MSHRs), and hardware prefetchers [10]. It incorporates a small speculative cache (GM) to store data linked to speculative loads. When a load is confirmed, the data is then propagated to the entire cache hierarchy, including the first-level data cache (L1D), second-level cache (L2), and the last-level cache (LLC). Importantly, GM is neither inclusive nor exclusive to the rest of the cache hierarchy.

On average, GhostMinion introduces a performance loss of approximately 5% compared to a non-secure cache system. This trade-off is deemed acceptable given the heightened security measures it provides against speculative execution attacks.

Data prefetchers play a crucial role in enhancing cache performance by turning cache misses into hits. Recent advancements in data prefetchers have significantly boosted single-thread performance, achieving average performance gains of 3

However, there's a downside to hardware prefetchers, even in secure cache systems. Prefetchers, trained and triggered on speculative loads, can become a potential

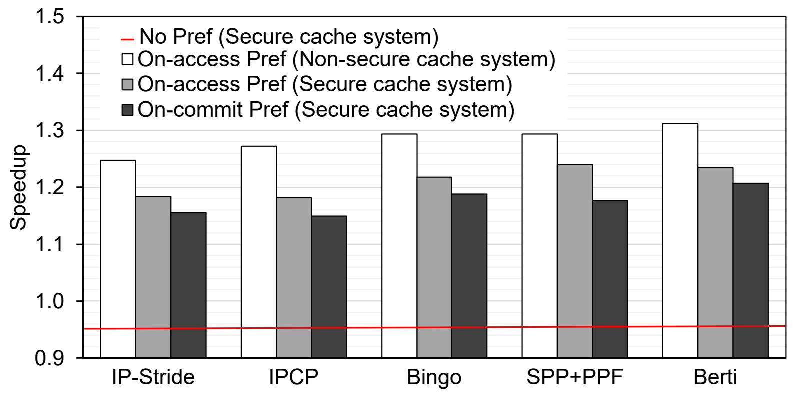


Figure 1.1: Speedup of state-of-the-art prefetchers

source of information leakage. A speculative attack leveraging prefetchers unfolds as follows [10; 11]: (i) The attacker primes the cache with a corresponding prefetcher; (ii) The victim loads secret data, akin to the Spectre attack; (iii) The speculative load by the victim trains and triggers the hardware prefetcher; (iv) The prefetcher requests data based on its prediction, affecting the cache state; (v) Finally, the attacker probes the cache.

GhostMinion argues for secure hardware prefetching through *on-commit* prefetching. This means a secure prefetcher should be trained and triggered only upon commit. This approach prevents the prefetcher from speculatively affecting the cache and MSHR (miss status holding registers) state, making it resistant to exploitation by transient instructions for information leakage. Our work contends that secure data prefetching can mitigate the performance loss of a secure cache system.

Despite the significance of data prefetching, there has been no comprehensive study on the impact of secure prefetching techniques. This work conducts a pioneering analysis of a range of state-of-the-art hardware prefetchers—IP-stride [12], the industry-standard prefetcher, Bingo [13], SPP+PPF [16], IPCP [31], the winner of the 3rd data prefetching championship [5], and Berti [30]—on secure cache systems. The analysis identifies two primary factors limiting their optimal performance and proposes microarchitectural solutions to address these limitations.

Our observations. Initially, we assess the effectiveness of the studied prefetchers in both non-secure and secure cache environments, utilizing SPEC CPU 2017 and GAP workloads. In Fig. 1.1, the speedup achieved by the prefetchers is compared to a non-secure cache system without prefetching (baseline). The red line indicates the relative performance degradation (5.1%) experienced by GhostMinion without a prefetcher. The first bar illustrates the performance enhancement in our non-secure baseline.

The second bar showcases the corresponding improvement in a secure cache system with standard on-access prefetching. Our initial observation reveals that none of the evaluated prefetchers narrows the performance gap between secure and non-secure systems. Although prefetching boosts performance for both, the gap widens. For instance, the advanced Berti prefetcher exhibits a 7.9% loss with on-commit prefetching in the GhostMinion system compared to a non-secure setup. Further analysis indicates that the increased memory latency, stemming from additional memory traffic due to on-commit data requests to L1D, L2, and LLC, is a primary factor. On average, GhostMinion introduces over 1.5× additional traffic to L1D compared to a non-secure system with hardware prefetchers.

Subsequently, we investigate the impact of implementing a secure prefetcher in a secure cache system—specifically, the effect of conducting training and prefetching during commit rather than cache accesses. Fig.1.1 illustrates the performance improvements for our studied prefetchers in a secure cache system when trained and triggered on access (second bar) and on commit (third bar). Our second observation is a consistent performance loss of 3%-4% for all prefetchers when shifting training/prefetching to the commit stage compared to on-access prefetching. Further exploration reveals that the primary factor is timeliness, not the inability to capture application access patterns. A significant portion of the performance loss for on-commit prefetching is attributed to a new category of late prefetch requests termed "commit-late": misses whose prefetching hadn't commenced when the processor requested the data, but would have initiated if triggered on access.

This work delves into the factors contributing to the suboptimal performance of secure prefetchers and proposes a cost-effective solution to recover this performance loss. The aim is to unlock the full potential of secure prefetching, bridging the performance gap with non-secure cache systems.

Our approach. Initially, we address the challenge of heightened memory hierarchy traffic in a secure cache system by implementing a filtering mechanism. This mechanism filters out non-speculative (on-commit) updates when they are unnecessary, resulting in a consistent performance improvement for all evaluated prefetchers.

Subsequently, we introduce an effective mechanism tailored for the leading prefetching mechanism, Berti. This mechanism ensures timely prefetch requests during commit by adjusting the latency used during training. The outcome is a secure, high-performing, and timely prefetcher.

This work brings the following contributions:

- We demonstrate that prefetchers experience a decline in relative performance in a secure cache system, attributed to both the augmented memory traffic introduced by the secure cache system and issues related to the timeliness of prefetching.

- We suggest an economical mechanism to filter out non-speculative updates in the memory hierarchy of a secure cache system. Our lightweight filter introduces minimal storage overhead, amounting to 0.12KB .

- We present a cost-effective mechanism to enhance the timeliness of a prefetcher trained to issue prefetch requests during commit. This mechanism advocates for a secure, timely, and high-performing prefetcher. The ultimate outcome is the development of the first high-performance secure prefetcher with a minimal storage overhead of 0.47KB.

- We demonstrate that the integration of our proposed modifications effectively narrows the performance gap between a non-secure cache system and a secure cache system, particularly leveraging the state-of-the-art Berti prefetcher. For SPEC CPU 2017 and GAP benchmarks, our modifications yield a performance improvement of 6.3% (1.9% from the filter and 4.4% from the enhanced on-commit prefetcher). In a 4-core system, our mechanisms enhance performance by 23.0% compared to the state-of-the-art Berti in a secure system.

Background

2.1 Threat Model

We make the following assumptions about the capabilities of our transient execution attacker:

(i) The attacker is proficient in executing attacks similar to Spectre, utilizing a cache state. Additionally, the attacker can carry out speculative interference attacks[15] through the cache system. It is noteworthy that we do not concentrate on transient fetch/decode attacks, such as Phantom [45], as they impact instruction prefetchers without affecting data prefetchers.

(ii) The attacker has the ability to exploit a hardware prefetcher through speculative training and prefetching, thereby altering the cache state, as described in Muontrap [10].

(iii) The attacker and the victim operate as two distinct processes. The attacker can execute arbitrary code but is unable to directly access secret data. In other words, the attacker operates within a sandbox, either at the user or kernel level.

(iv) While there are other microarchitecture attacks involving hardware prefetchers, like AfterImage [19], which intentionally train and trigger prefetching across domain switching, and data memory-dependent prefetchers that pose threats like Augury [43], classical state-of-the-art prefetchers are unaffected. Timing-based side and covert channels involving hardware data prefetchers and caches [40; 20] can be mitigated by existing spatial or temporal isolation techniques [22; 34]. For prefetchers, flushing of prefetch tables or spatial/temporal partitioning of prefetch tables across different domains can mitigate non-speculative channels. Hence, our focus is solely on transient attacks involving a cache system.

2.2 Hardware prefetchers

Cutting-edge data prefetchers have significantly enhanced single-thread performance, achieving average performance boosts ranging from 3% to 5% [16; 13; 31; 30]. The

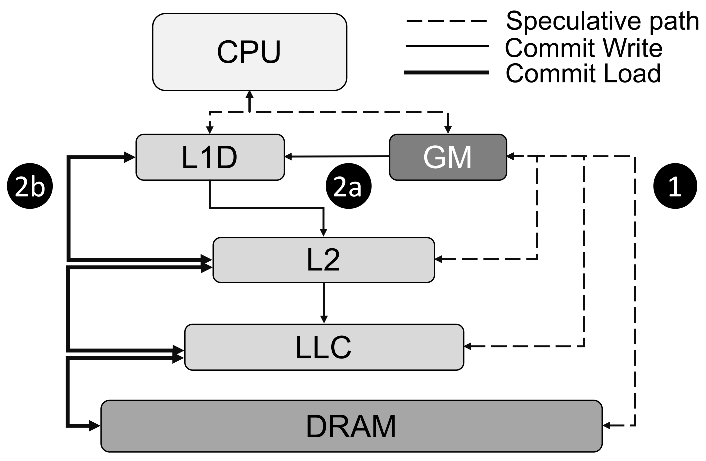


Figure 2.1: GhostMinion secure cache system

majority of recently proposed storage-efficient prefetchers are designed for the L2 cache [13; 16]. Exceptions include multi-lookahead offset prefetching (MLOP) [38], instruction pointer classifier-based prefetching (IPCP) [31], and Berti [30], which are L1D prefetchers. It is well-established that an L1D prefetcher offers superior performance compared to an L2 prefetcher because prefetched lines are brought closer to the core, as demonstrated by IPCP [31]. Additionally, an L1D prefetcher observes unfiltered memory access patterns, enabling more accurate predictions of future accesses compared to an L2 or LLC prefetcher. L1D also processes a sequence of virtual addresses, facilitating cross-page prefetching [25]. Moreover, contextual information, such as the instruction pointer (IP), is more readily available at L1D (e.g., Intel's IP-stride at L1D [23]) than at L2 and LLC. Berti stands out as the state-of-the-art L1D prefetcher, boasting high accuracy of almost 90%. It orchestrates requests across the entire cache hierarchy without polluting L1D or wasting memory hierarchy bandwidth. By distributing requests to various levels of the hierarchy (L1D, L2, LLC), Berti alleviates pressure on critical structures at the first level, such as the prefetch queue (PQ) and Miss-status holding registers (MSHRs). Notably, Berti incurs a minimal storage overhead of only 2.55 KB. Throughout the paper, we concentrate on Berti as the chosen L1D prefetcher, given its superior performance compared to IP-stride and IPCP.

2.3 Secure cache system: GhostMinion

GhostMinion employs a compact 2KB cache named GM, accessed concurrently with the L1D, designed to store speculative instruction data until they commit or retire. In the event of a demand miss initiated by a speculative instruction in GM, the system searches for the data in L1D, L2, and LLC, mimicking a traditional cache hierarchy. However, if there is a hit in L1D, L2, or LLC, the cache state, specifically the replacement policy priority bits, remains unaltered. In the case of a miss in L1D, L2, and LLC, the response is directly filled into GM, bypassing L1D, L2, and LLC (see Fig. 2.1, 1). Upon commit, the data of committed instructions from GM is transmitted to the rest of the cache hierarchy (L1D, L2, and LLC) if it is a GM hit, accomplished through on-commit writes (see Fig. 2.1, 2a). If there is a GM miss, a re-fetching of data is conducted into the non-speculative cache hierarchy (L1 to LLC) (see Fig. 2.1, 2b). Importantly, GM is neither inclusive nor exclusive to the remainder of the cache hierarchy. Instructions within GM are constrained from observing the eviction or insertion of others based on their temporal order, maintained through timestamps. TimeGuarding, a technique employed by GhostMinion, ensures the invisibility of data and evictions under multiple speculations. To conceal contention at various queues, timestamp metadata propagates into the Miss-status holding registers (MSHRs) at each cache level, enabling the cancellation and replacement (leapfrogging) of loads.

Motivation

3.1 Impact of secure cache system on hardware prefetching

This section scrutinizes the factors impeding the efficacy of prefetchers on secure cache systems like GhostMinion, identifying the primary cause as the traffic generated during the restoration of the cache state on commit. Fig. 3.1 illustrates the rise in L1D accesses per kilo instructions (APKI) for the assessed prefetchers in a GhostMinion secure cache system and for on-access prefetching. In a non-secure system without prefetching, the average APKI is 199, escalating to 375 in GhostMinion due to commit requests updating the cache state. This trend persists across all prefetchers. For L2 prefetchers such as Bingo and SPP+PPF, there is no access from the prefetcher to L1D, as prefetch requests are generated from L2.

The increase in APKI results in additional traffic, leading to an augmentation in L1D miss latency, as depicted in Fig. 3.2. A significant contributor to this increased miss latency is an intriguing trend, particularly noticeable in the presence of hardware prefetching. On average, for the Berti prefetcher in a secure cache system, there is a 10% increase in L1D Miss Status Holding Register (MSHR) occupancy, and the L1D MSHR becomes full for an additional 8.7% of the time. Furthermore, without prefetching, L1D MSHR occupancy decreases by 16% when transitioning from a non-secure to a secure cache system, as demand misses are initially handled by GM. However, with prefetching, there is a 10% increase in MSHR occupancy when moving from a non-secure to a secure cache system.

To delve deeper into these interactions, we select the `605.mcf_s-1554B` workload for detailed analysis. Fig. 3.3(a) shows the normalized performance relative to a non-secure baseline without prefetching, revealing a substantial performance reduction when applying prefetchers to a secure cache system (over 300% for Berti). Fig. 3.3(b) illustrates

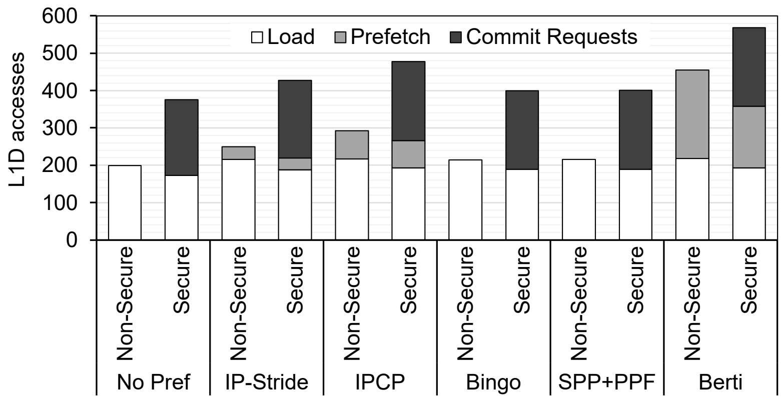


Figure 3.1: Average L1D accesses per kilo instruction with on-access prefetching

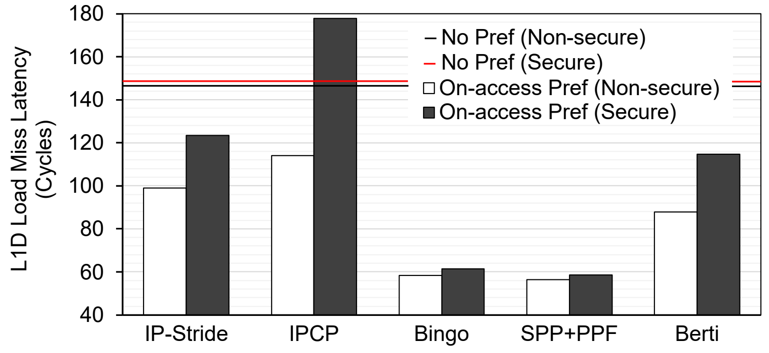


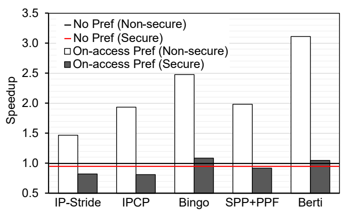
Figure 3.2: Average L1D load miss latency with on-access prefetching

the increased traffic at L1D caused by load, prefetch, or commit requests from GhostMinion. Fig. 3.3(c) demonstrates a significant rise in L1D miss latency.

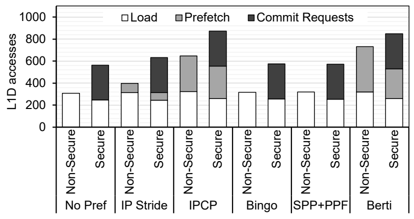
When scrutinizing MSHR occupancy without prefetching, L1D MSHR occupancy decreases by 16% when transitioning from a non-secure to a secure cache system, as demand requests are initially handled by GM. However, with prefetching, there is a 10% increase in L1D MSHR occupancy when moving from a non-secure to a secure cache system. This is attributed to the fact that, in a non-secure cache system, the L1D MSHR deals only with demand and prefetch requests, while in a secure cache system, it also has to handle GhostMinion requests, intensifying the pressure on the MSHR. Without prefetching, L1D MSHR is almost never full, but with prefetching, there is an increase in the percentage of time L1D MSHR is full (from 6.3% to 20%). We introduce a mechanism to alleviate these additional traffic-induced performance losses when hardware prefetching is enabled.

3.2 Impact of secure hardware prefetching

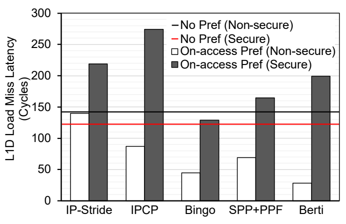
In GhostMinion [11], on-commit prefetching ensures the non-leakage of information during speculative execution. However, as indicated in Fig. 1.1, the straightforward transition



(a) Normalized speedup



(b) L1D traffic



(c) L1D Load Miss Latency

Figure 3.3: Speedup normalized to a non-secure cache system with no prefetching, traffic (in terms of APKI), and miss latency with on-access prefetching for 605.mcf_s-1554B trace

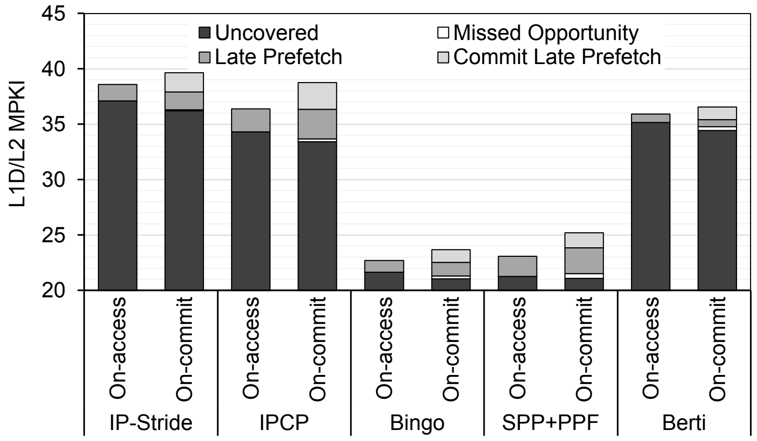


Figure 3.4: Average L1D/L2 demand MPKI in terms of coverage and lateness.

of state-of-the-art prefetchers to the commit stage results in a 3%-4% performance loss compared to on-access prefetching. This section investigates the reasons behind this observation.

Fig. 3.4 presents the average demand misses per kilo instructions (MPKI) across the analyzed workloads. The MPKI is shown for the cache level where the prefetcher operates, specifically L1D for IP-stride, IPCP, and Berti, and L2C for Bingo and SPP+PPF. Each prefetcher is evaluated with both on-access and on-commit prefetching. The MPKI is categorized into the following four groups:

- *Commit late prefetch*: This is a new kind of late prefetch request that we introduce in this work and it only appears when the prefetcher is placed at the commit stage. We define it as follows: at the time of a demand cache miss, a prefetch request for the target cache line has not been triggered yet by the on-commit prefetcher, but it would have been triggered by an on-access prefetcher. Importantly, this type of prefetch does not fall in the traditional late prefetch category since in fact the prefetcher request has not been triggered yet when the access takes place.

- *Late prefetch*: This is the typical late prefetch, where a demand miss finds in the MSHR a prefetch request for the target cache line, and merges both requests.
- *Missed opportunity*: The demand miss is for a cache line that would have been predicted correctly by an on-access prefetch but it was missed by on-commit prefetch as it is trained in a different order. This kind of prefetch is also only present for on-commit prefetching and gives information about the negative impact on address prediction of training at commit.
- *Uncovered*: Demand misses that did not fall in any of the previous categories.

We observe a common trend for all evaluated prefetchers: the uncovered demand misses are reduced when moving the prefetcher to on-commit. Even if we add the missing opportunity bar to the uncovered one, in general, the resulting MPKI is lower for on-commit prefetchers. This demonstrates that on commit, address prediction is more effective than on access. This is an expected result since cache accesses are commonly reordered by the out-of-order engine and that order may vary from iteration to iteration. However, on commit, instructions always train the prefetcher in the same order.

Despite the advantage on prediction of on-commit prefetching, performance is worse when compared to on-access prefetching. The reason is timeliness. Although traditional late prefetch requests practically do not increase when moving from on-access to on-commit, our new defined class of *commit late* is the culprit of the increase in overall MPKI for on-commit prefetching. That is, prefetch requests need to be triggered earlier to compensate for the delays entailed by on-commit prefetching. Fortunately, as we show in this work, it is possible to compensate for this lack of timeliness. Chapter 5 proposes a mechanism that mitigates the lack of timeliness.

Prefetch-friendly secure cache system

Secure cache systems update the cache hierarchy when memory instructions are non-speculative, such as during the commit stage. This involves either re-fetching the cache line from the GM on a miss or propagating on-commit write requests (for clean cache lines) to the rest of the cache hierarchy on a hit in the GM. The purpose of this additional data movement is to populate the cache hierarchy, which remained unchanged when the data was speculatively requested by the core, in order to minimize cache misses in subsequent accesses.

While this extra traffic typically does not lead to noticeable performance degradation in an uncongested memory system without prefetching mechanisms, it becomes apparent when prefetchers stress the cache hierarchy queues and MSHRs, hindering their ability to enhance performance.

Our observation reveals that many requests made to restore the cache hierarchy are unnecessary and contribute to significant contention within the cache hierarchy. For instance, initiating a re-fetch for data already provided by the L1D would consume L1D ports solely to update the LRU replacement policy. Similarly, on-commit write requests propagate up the memory hierarchy until the data is encountered in a cache level, making access to the cache level containing the cache line redundant.

Based on these insights, we introduce the secure update filter (SUF). SUF keeps track of the cache level that provided the data when requested. Then, at commit time, SUF either filters the re-fetching when the data originated from the L1D or halts the on-commit write propagation at the level preceding the one that furnished the cache line. While there is a risk of misprediction if the fetched cache line has been evicted in the interim, SUF's high accuracy minimizes the number of cache accesses and, consequently, the traffic generated during cache state restoration. SUF operates independently of the underlying prefetcher mechanism in a transparent manner. The subsequent section outlines the detailed implementation of SUF, depicted in Fig. 4.1.

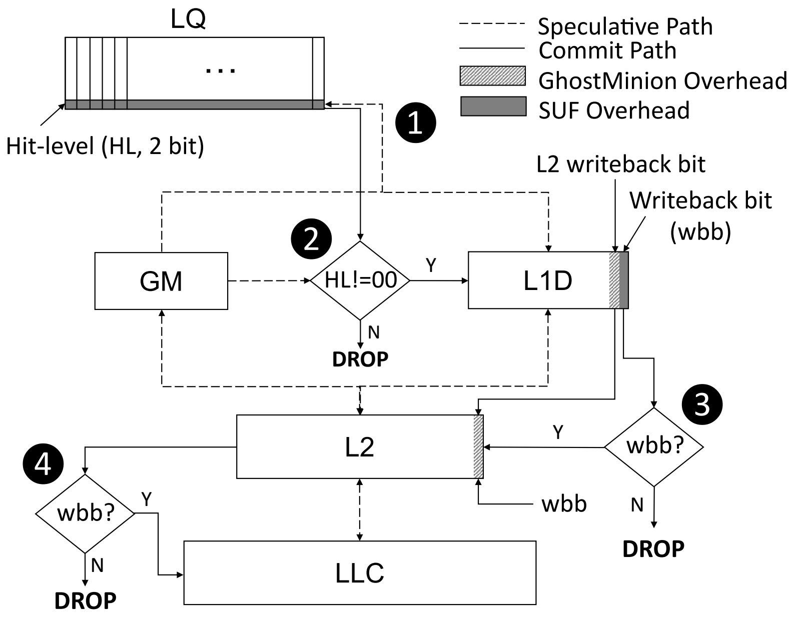


Figure 4.1: Overview of the secure update filter (SUF)

Identifying the Cache Level Holding a Cache Line. SUF determines the lower level (with L1D being the lowest and LLC the highest level of the cache) holding a cache line to decide whether filtering should be applied. The cache level is identified when the processor requests the data by propagating down the hierarchy the cache level that served the cache line. This information is encoded using 2 bits, indicating if the data comes from L1D (or GM, accessed in parallel), L2C, LLC, or DRAM. The 2-bit hit-level information is stored alongside the requested data in the memory operation entry at the load queue (LQ) (Fig. 4.1, 1).

Filtering updates. After the speculative load instruction is committed, it consults the GM to determine whether to re-fetch the cache line down the hierarchy (GM miss) or propagate the cache line up the hierarchy (GM hit). SUF examines the hit-level field and takes the following actions. In the case of the data being provided by the L1D (value 00), SUF cancels the update (both for re-fetching and on-commit propagation) (Fig. 4.1, 2). Otherwise, the re-fetch proceeds as usual, or propagation causes the cache line to move from GM to L1D. In addition to a *writeback bit*[1] used in GhostMinion, we utilize an additional L2 writeback bit that determines whether to extend the on-commit write chain beyond L2. Upon eviction of the cache line from either L1D or L2C, the decision to propagate the writeback block is determined by the GhostMinion writeback bit, and the L2 writeback bit is mirrored to the L2 cache (Fig. 4.1, 3). Finally, during eviction from L2, the GhostMinion writeback bit is once again employed to decide whether to propagate the cache line or not (Fig. 4.1, 4).

[1]While the GhostMinion paper doesn't explicitly mention this bit, we assume its implicit presence.

Storage overhead. SUF is deployed with a minimal storage overhead of only 0.12 KB: 0.03 KB in the LQ and 0.09 KB in the L1D. Specifically, each of the 128 entries in the LQ is expanded with a two-bit hit-level field, and each of the 768 entries in the L1D is extended with a single L2 writeback bit.

Timely secure prefetcher

Transitioning prefetchers from on-access to on-commit results in an average performance decline of 3% to 4%. This reduction in performance is attributed to the new timeliness constraints, introducing *commit-late* prefetch requests and missed prefetch opportunities. These factors collectively contribute to a decrease in prefetch accuracy and coverage.

We observe that this behavior can be rectified by adjusting prefetch timeliness. One approach to improve prefetch timeliness is to increase the prefetch distance, covering more distant memory requests and consequently reducing commit-late prefetch requests. However, increasing the prefetch distance also raises the pressure on queues and MSHRs, necessitating a new mechanism to dynamically control the prefetch degree [41].

Another option to address the timeliness issue is to modify the learning process of the hardware prefetcher. In this regard, we focus on Berti. Berti is chosen because (i) it exhibits the most significant performance improvements among its counterparts, and (ii) its learning mechanism is closely linked to the timing of prefetch and demand requests, making it well-suited for adaptation to latency changes. To comprehend our mechanism for effectively adapting Berti's latency training in a secure cache system, we provide the necessary background on the Berti prefetcher.

5.0.1 Berti prefetcher: 10K feet view

Prefetcher Training Process. Berti undergoes training for deltas associated with a specific Instruction Pointer (IP), referred to as local deltas. Local deltas represent the difference between the cache line addresses of two demand accesses. The training mechanism aims to estimate the coverage of each observed delta per IP, focusing only on those deltas conducive to timely prefetching. The training process encompasses three key steps: measuring fetch latency, learning timely deltas, and computing the coverage of the deltas.

1. Measuring fetch latency. To identify timely deltas, it is essential to measure the time required to fetch data to the L1D, denoted as L1D miss latency. This measurement includes both demand misses and prefetch requests for any cache line in L1D. The fetch latency is tracked by assigning a timestamp to each L1D miss in the MSHR and each prefetch request in the PQ. Upon L1D fill, the latency is calculated by subtracting the stored timestamp from the current one.

2. Learning timely and accurate deltas. With fetch latency data available for each L1D fill, Berti precisely learns timely deltas based on the history of accesses and recorded timestamps by the same IP. Deltas are computed by subtracting the address of each timely request in the history from the current address.

3. Computing the coverage of deltas. During each search in history, Berti retrieves a set of timely deltas. Deltas that frequently appear in these searches contribute to high coverage, while rarely occurring deltas result in low coverage. The coverage is computed by dividing the number of occurrences of a delta by the number of searches in the history. Local coverage, specific to each IP, correlates with accuracy. A delta covering 100% of cache lines, for instance, achieves 100% accuracy as each access-delta pair generates only one prefetch request.

Prefetch Request Issuance. Once Berti determines the deltas and their associated coverage, it coordinates prefetch requests across the cache hierarchy. If a delta's coverage exceeds a *high-coverage* watermark and the L1D MSHR occupancy is below the *occupancy* watermark, prefetch requests employing that delta are filled at all cache levels up to L1D. Conversely, if the coverage surpasses a *medium-coverage* watermark, prefetch requests are filled up to L2, irrespective of the L1D MSHR occupancy.

5.0.2 Issues with the secure prefetcher

As discussed in the previous section, the functioning of Berti relies on fetch latency for its learning mechanism. However, when adapting Berti for secure prefetching, two primary challenges emerge: (i) the L1D observes a deceptive on-commit latency instead of the actual on-access latency, and (ii) the optimal trigger time for prefetch requests should be relative to commit events rather than access events.

To delve deeper into these challenges, let's first examine the on-commit version of Berti integrated into GhostMinion. In this context, Berti, situated at the L1D cache, utilizes post-commit L1D accesses and fills for training. GhostMinion speculatively fills the GM and transfers the cache line to L1D upon commit. Consequently, Berti perceives the on-commit write latency from GM to L1D rather than the fetch latency from a higher memory hierarchy level to GM. This shift in fetch latency disrupts the learning process, leading to inaccuracies in delta learning and subsequent prefetch requests, ultimately resulting in performance degradation. Figure 5.1 visually illustrates the training and prefetch request issuance process for on-commit Berti in GhostMinion. The timeline

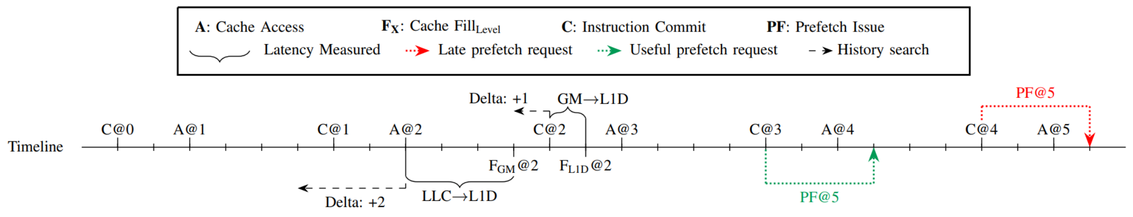


Figure 5.1: On-commit Berti (above) and TSB (below) of the timeline. The timeline represents the access/commit of an IP, with each step representing one cycle.

depicts a +1 delta for a given load, with all L1D accesses resulting in misses that take three cycles to fill GM and one cycle for on-commit writes from GM to L1D.

Upon the commit of the load for @2 (C@2), the learning process is initiated. When cache line @2 fills the L1D cache from GM through on-commit write, the latency of one cycle is recorded. Subsequently, Berti searches for the nearest commit instruction capable of triggering a prefetch request that aligns with the recorded latency. In this example, this instruction is C@1. Berti then learns the appropriate delta, considering the writeback latency, which, in this case, is +1. From this point onward, Berti triggers a prefetch request with delta +1 upon each instruction commit. When the load for @4 is committed (C@4), Berti issues a prefetch request for line@5 (PF@5). However, this prefetch request takes three cycles to fill the cache, resulting in a delayed prefetch request since the access to cache line @5 is performed two cycles later. This delay arises because (i) the learning process is based on writeback latency rather than fetch latency, and (ii) it considers when the access to that cache line was performed (A@2) rather than when it was committed (C@2). It's important to note that this late prefetching issue persists even if Berti searches for deltas based on the cache access time (A@2) rather than the commit time (C@2). Thus, both challenges must be addressed to achieve timely secure prefetching.

5.0.3 Timely training of the secure prefetcher

Motivated by the insights gained from the preceding observations, we introduce Timely Secure Berti (TSB), a novel training mechanism for Berti that leverages fetch latency to GhostMinion (GM) and computes deltas based on access times. This approach emulates the latency that future demand accesses will encounter, addressing the challenges associated with on-commit prefetching.

The operation of TSB is as follows: when a demand load miss occurs, TSB speculatively captures essential information for accurate Berti training, including the demand time and the fetch latency to GM. Upon the commitment of the memory instruction, TSB utilizes the stored information to update its history (recording the demand miss time as the

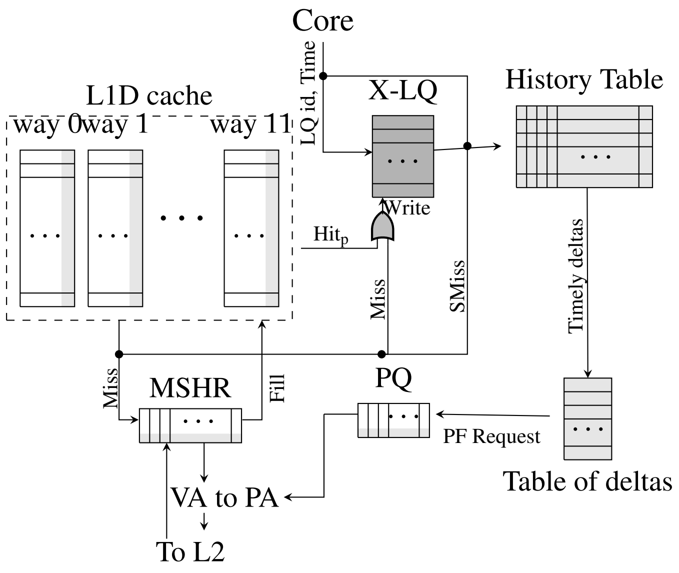


Figure 5.2: TSB design overview. Berti original hardware extensions are shown in light gray. TSB hardware extensions are shown in gray.

history timestamp) and identify deltas (using the fetch latency to GM). This mechanism enables TSB to learn timely deltas.

As illustrated in Fig. 5.1 (below the timeline), when cache line @2 fills the GM, the fill latency is computed as the difference between the fill timestamp and the cache access timestamp, saved until commit. Upon the commitment of instruction @2 (C@2) and the readiness of Berti for training, the fill latency is used to identify the commit that can prompt a timely prefetch request aligning with the access of cache line @2. Once the commit is identified (in our example, C@0), the delta is calculated (+2). In contrast to the default on-commit version, TSB triggers prefetch requests with delta +2. In our example, C@3 triggers the prefetch request for cache line @5, resulting in a timely prefetch request.

Hardware Implementation: TSB can be implemented with a hardware overhead of only 0.47 KB over Berti (3.01 KB over no-prefetch). Fig. 5.2 provides an overview of TSB, where light gray represents the additional Berti hardware, and dark gray represents the extra TSB hardware.

In essence, TSB incorporates an Extended Load Queue (*X-LQ*), an extension of the Load Queue (LQ) required to convey the actual fetch latency information.

The *X-LQ* consists of the same number of entries as the LQ (128 entries in our modeled system) and maintains a one-to-one mapping with LQ entry IDs. Each entry in the X-LQ includes a valid bit, a bit indicating a hit on a prefetched cache line (Hit$_p$), a 16-bit access timestamp, and a 12-bit fetch latency.

Upon an L1D miss, the valid bit is set, and the access timestamp is populated with information from the local processor's clock (the last 16 bits of the current cycle). When the cache fill in the GM is completed, the fetch latency is recorded. In the case of a hit on a prefetched cache line, both the valid bit and the Hit_p bit are set. Here, the access timestamp is also recorded, while the fetch latency corresponds to the prefetched line's latency, previously computed and stored alongside the L1D cache [30]. For a regular hit, the valid bit remains unset, as no action is taken at the commit stage.

Upon the commitment of a load, the history table is populated for both misses and prefetch hits using the access timestamp from the X-LQ (rather than the commit event time), and the deltas are sought using the X-LQ fetch latency.

Evaluation

6.1 Methodology

The evaluation in this study utilizes ChampSim [7], a trace-driven simulator that was employed for the 2nd and 3rd Data Prefetching Championships (DPC-2 [1] and DPC-3 [5]). Recent prefetching proposals have also been implemented and assessed on ChampSim [16; 13; 31; 38; 30]. The version utilized in DPC-3 has been extended with a decoupled front-end [33], comprehensive support for the memory hierarchy in terms of address translation, and an accurate DRAM model that considers various factors affecting access time, such as bank conflicts, close/open page policies, page hits/misses, and more.

Dynamic energy consumption for the memory hierarchy, encompassing caches and DRAM, is calculated using CACTI-P [29] and the Micron DRAM [2] power calculator, both based on a 7 nm process technology. Table 6.1 provides detailed information about our baseline system configuration, resembling an Intel Sunny Cove microarchitecture [24; 3; 4].

We utilize publicly available traces [6; 8] sourced from the SPEC CPU2017 [42] and single-threaded GAP [14] benchmark suites. Our study focuses on the 65 memory-intensive traces (45 from SPEC CPU2017 and all from GAP) that exhibit at least one miss per kilo-instruction (MPKI) at the Last-Level Cache (LLC) in our baseline system. Simulations are conducted for both single-core and multi-core scenarios. We collect statistics over 200 million simulated instructions after a warm-up period of 50 million instructions [39]. In multi-core experiments, we simulate 150 randomly generated heterogeneous mixes of SPEC CPU2017 and GAP traces and report the weighted speedup.

The effectiveness of Secure Update Filter (SUF) and Timely Secure Berti (TSB) is evaluated on a GhostMinion [11] secure cache system featuring a 2KB, 1-cycle Ghost Minion (GM). The evaluation includes a comparison with state-of-the-art data prefetchers: IP-Stride [23] (the Intel and AMD L1D prefetcher), IPCP [31] (the winner of the

Table 6.1: Simulation parameters of the baseline system.

Core	Out-of-order, hashed perceptron branch predictor [26], 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle STLB: 1536 entries, 12-way, 8 cycles
L1I	32 KB, 8-way, 4 cycles, 8 MSHRs, LRU
L1D	48 KB, 12-way, 5 cycles, 16 MSHRs, LRU
L2	512 KB, 8-way, 15 cycles, 32 MSHRs, LRU, non-inclusive
LLC	1 bank per core, each bank: 2 MB, 16-way, 35 cycles, 64 MSHRs, LRU, non-inclusive
DRAM	Controller: One channel/4-cores, 6400 MTPS [21], FR-FCFS, reads prioritized over writes, write watermark: 7/8th Chip: 4 KB row-buffer per bank, open page, burst length 16, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

DPC-3 competition), Bingo [13], SPP+PPF [16], and Berti [30]. Each prefetcher is implemented with highly tuned parameters, as detailed in Table 6.2.

6.2 Results

This section presents a thorough evaluation of state-of-the-art prefetching techniques in secure systems and shows the benefits that can be obtained when using our two main contributions: the secure update filter (SUF) and the timely secure Berti (TSB) prefetcher.

All normalized graphs are relative to a non-secure system without prefetching. If a red line is present, it represents a GhostMinion secure cache system without prefetching. When averaging results, we use the geometric mean when normalizing values and the arithmetic mean otherwise. In graphs showing average numbers, each bar represents a prefetch configuration: on-access prefetch in a non-secure cache system (white bar), on-commit prefetch in a GhostMinion cache system (gray bar), and on-commit prefetch in a GhostMinion system with the SUF mechanism (black bar). The last prefetcher, TSB, is our timely secure prefetcher.

6.2.1 Performance

Overall speedup. The average single-thread speedup of the prefetchers evaluated for both non-secure and secure systems is shown in Fig. 6.1. The first (while) bar shows the speedup of the non-secure version of the prefetchers. The second (gray) bar shows the speedup obtained in a GhostMinion secure cache system by the same prefetchers, now

Table 6.2: Configurations of evaluated prefetchers

Prefetcher	**Configuration**	**Size**
IP-Stride	1024 entries	8KB
IPCP [31]	128-entry IP table, 8-entry RST table, and 128-entry CSPT table	0.87KB
SPP+PPF [16]	256-entry ST, 512-entry 4-way PT, 8-entry GHR, Perceptron weights: 4096×4, 2048×2, 1024×2, and 128×1 entries, 1024-entry prefetch table, 1024-entry reject table	39.2 KB
Berti [30]	128-entry History Table, 16-entry Delta table with 16 deltas	2.55 KB
Bingo [13]	2 KB region, 64/128/16K-entry FT/AT/PHT	124 KB

being secure. All prefetchers exhibit a performance loss (between 7.3% and 9.6%) when transitioning from non-secure to secure, in part due to the 5.1% performance degradation of GhostMinion (red line). The third (black) bar shows the speedup achieved by SUF, which improves the performance of all secure prefetchers with the highest improvement of 3.7% for Bingo and the lowest of 1.9% for Berti.

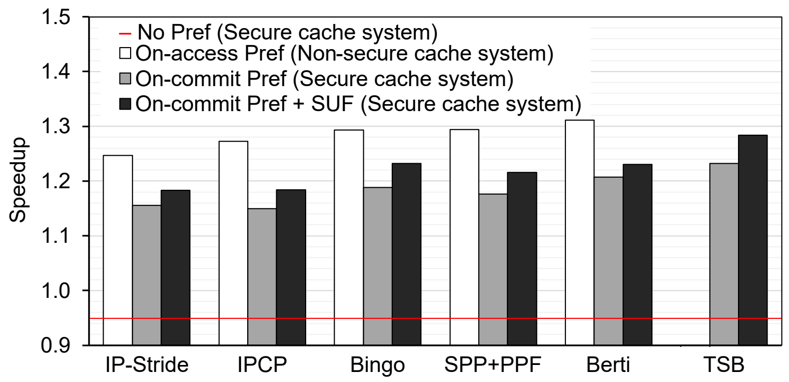
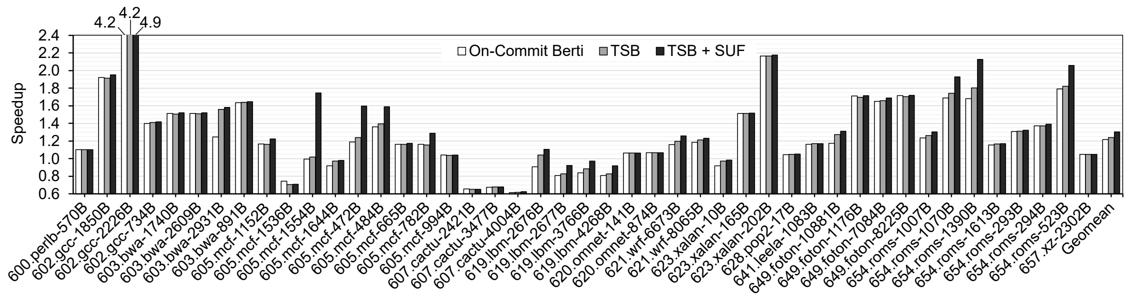
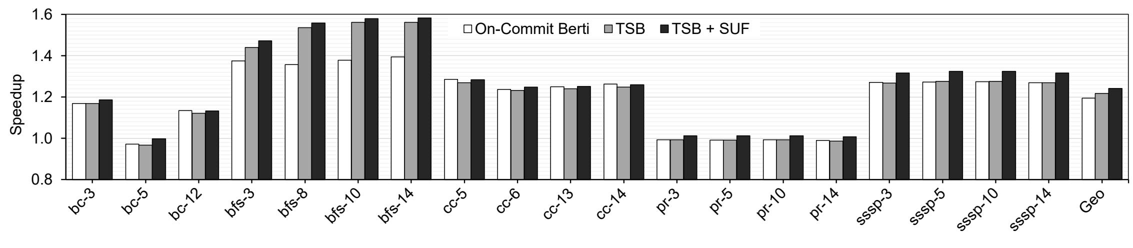


Figure 6.1: Speedup normalized to non-secure cache system with no prefetching. The higher the better.

The last set of bars illustrates the improvements of our TSB proposal, only for secure scenarios. TSB without SUF achieves the same speedup as secure Berti + SUF (23%). When SUF is added to TSB, the speedup increases by 4.2%, reaching a total of 28.4% (over baseline). Among the on-access prefetchers in a non-secure system, only Berti offers a clear performance advantage over our secure TSB+SUF; SPP+PPF, Bingo, and IPCP offer similar performance (less than a 0.8% performance difference) and IP-Stride has lower performance (2.9%). Importantly, TSB+SUF mitigates the performance degra-



(a) SPEC CPU2017



(b) GAP

Figure 6.2: Speedup of Berti, TSB, and TSB+SUF normalized to a non-secure system without prefetcher. The higher the better.

dation of using a secure system from 5.1% (in a system without prefetching) to 2.1% (in a system with prefetching), a result that confirms our claim that prefetching techniques can alleviate the impact of security on performance.

Accuracy of SUF. On average SUF filters accurately for 99.3% of the time, with the maximum accuracy of 99.9% for `654.roms_s-1613B` and minimum of 87.26% for `605.mcf_s-1554B`, improving the effectiveness of GhostMinion with prefetching.

Individual speedup. Fig. 6.2 shows the individual speedup for on-commit Berti, TSB, and TSB+SUF. For SPEC traces, TSB improves by more than 5% in 7 out of 45 traces (15.6% of all traces). *603.bwaves_s-2931B* improves the performance by 24.9% over Berti, because it has a large fetch latency which is learned correctly in TSB. Our new learning system allows TSB to learn better and more accurate deltas, which provides better accuracy and coverage. TSB+SUF achieves more than 5% performance improvement in 18 out of 45 traces (40% of all traces), with a maximum improvement of 75.8% in *605.mcf_s-1554B*. After analyzing its behavior, we detected that this improvement comes from the reduction in the number of cycles that the L2 MSHR is found full by the application, which drops by 42.2%. TSB and TSB+SUF only sees a performance drop of more than 1% in one application, *605.mcf_s-1536B*. As for GAP, TSB achieves better performance in all *bfs* traces with an average improvement of 10.8%, also because of their large fetch latency. TSB+SUF achieves slightly better performance in all benchmarks, with more than 3.8% average performance improvement in *sssp* traces due to the inclusion

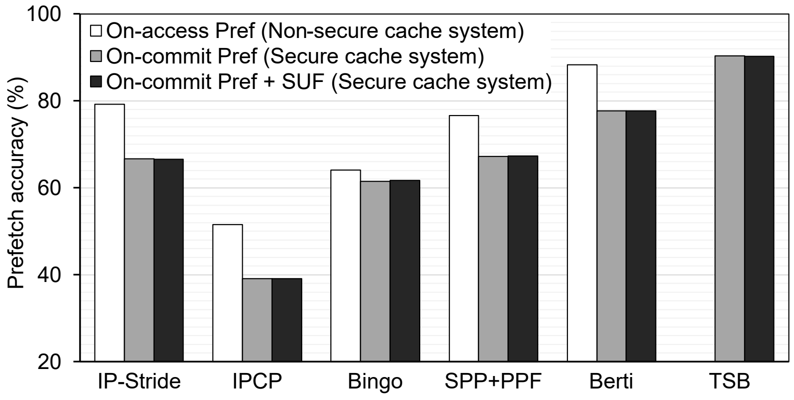


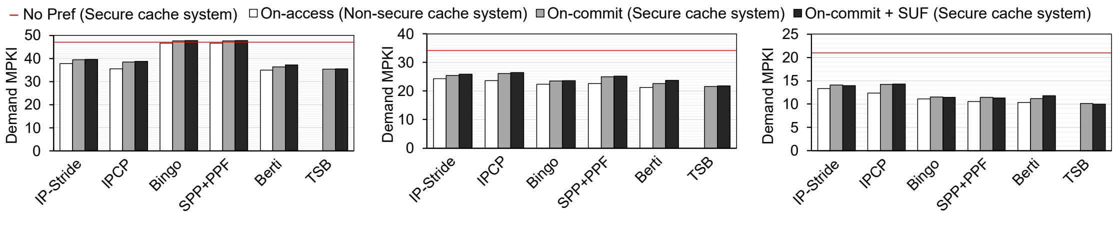
Figure 6.3: Average prefetch accuracy. The higher the better.

of the SUF filter. Interestingly, TSB and TSB+SUF do not degrade performance in any trace.

Sensitivity to GM size.. We analyze the performance sensitivity of different GMs (1 KB, 2 KB, and 4 KB). With all the different sizes, the single-thread performance improvement of SUF and TSB remains similar.

6.2.2 Accuracy and coverage

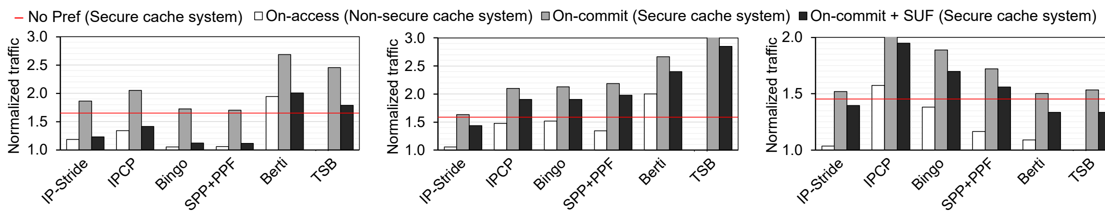
Accuracy. Fig. 6.3 shows the accuracy of the different prefetchers. Compared to the on-access prefetcher, on-commit prefetchers experienced a decrease in accuracy in all prefetchers, with a maximum of 24.0% in IPCP and a minimum of 4.1% in Bingo. Because SUF does not affect the timeliness of the prefetcher, it does not modify the accuracy of any prefetcher. The improvements in the learning system for TSB and TSB+SUF are reflected in their accuracy, which is 2.4% better than on-access Berti in non-secure systems, achieving an accuracy of 90.3%.



(a) L1D (b) L2 (c) LLC

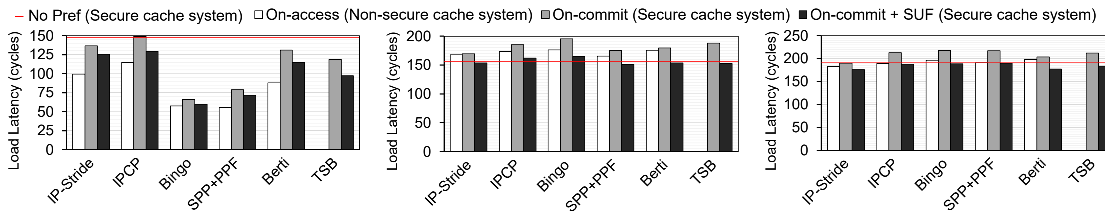
Figure 6.4: Average L1D, L2, and LLC demand MPKIs. The lower the better.

Coverage. Fig. 6.4 shows the demand misses per kilo instructions (MPKI) at the L1D, L2, and LLC (Y axis) with prefetchers (X axis). All prefetchers exhibit an average MPKI increase of 4.3%, 7.4%, and 8.3% at the L1D, L2, and LLC caches, respectively, when they move from on-access in a non-secure system to on-commit in a secure system. IPCP is the prefetcher that sees its coverage most affected, with a 3.9% increase in MPKI



(a) L1D (b) L2 (c) LLC

Figure 6.5: Normalized L1D, L2 and LLC traffic. The lower the better.



(a) L1D (b) L2 (c) LLC

Figure 6.6: Demand load latency in cycles at L1D, L2 and LLC for all prefetchers configuration. The lower the better.

at the L1D cache. As with accuracy, since SUF does not change the way that prefetchers work, its coverage remains the same for all of the prefetchers. Our secure prefetcher, TSB, achieves an MPKI reduction of 2.4%, 4.9%, and 8.8% at the L1D, L2, and LLC caches, respectively, compared to the second-best on-commit prefetcher (Berti). TSB and TSB+SUF have the same coverage as on-access Berti in a non-secure cache system. The superior coverage of TSB can be attributed to the higher latency seen by it, which provides the least late and incorrect prefetch requests.

6.2.3 Memory hierarchy traffic, latency and energy with SUF

Memory hierarchy traffic. GhostMinion adds a significant amount of traffic due to the writeback and re-fetch requests. Fig. 6.5 shows the traffic between the different levels of cache (Y-axis) under the different hardware prefetchers (X-axis). All prefetchers increase traffic compared to its on-access version by an average of 54.7%, 46.6%, and 40.4% in L1D, L2, and LLC, respectively. SUF is able to mitigate the increase in traffic generated by GhostMinion writeback and re-fetch actions with an active prefetch, for example, L1D prefetcher sees a reduction in traffic by an average of 30.1% on L1D.

Latency. Fig. 6.6 shows the number of cycles that load miss latency takes to fill the L1D, L2, and LLC caches (Y axis). The utilization of a secure system cache increases the latency of all prefetchers at all levels, with a significant increase at the L1D cache. On average, the on-commit prefetcher sees an increase in latency of 34.8% in the L1D cache,

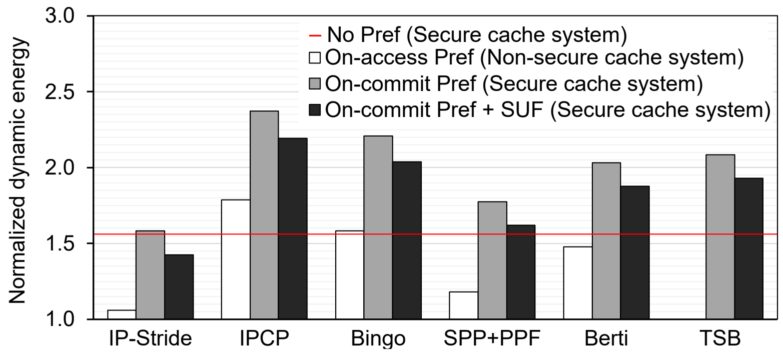


Figure 6.7: Normalized dynamic energy consumption. The lower the better.

5.2% in the L2 cache, and 8.8% in the LLC cache with respect to on-access prefetchers. Of all the prefetchers, Berti is the one whose latency is most affected, with a maximum increase of 49.4% in the L1D cache (from 87.8 cycles to 131.2 cycles). This is because Berti is the most aggressive of all the prefetchers, triggering more prefetch requests and increasing memory traffic. Thanks to the reduction in traffic between cache levels, SUF is able to reduce the latency penalty introduced by GhostMinion by more than 12% at all cache levels, which translates into higher performance.

Energy. Fig. 6.7 shows the normalized dynamic energy consumption in the memory hierarchy (L1D, L2C, LLC) (Y-axis) for the different prefetcher configurations (X-axis). There is a direct correlation between traffic and dynamic energy consumption overhead in the memory hierarchy. The secure system has extra traffic generated by GM, which increases the base energy consumption for all prefetchers. The on-commit version of the prefetcher increases energy consumption by an average of 41.8%, compared to the on-access version. SUF is able to reduce this increase in energy from 41.8% to 30.0%. On-commit IP-Stride+SUF is able to consume less than the system without prefetching, thanks to SUF reducing all the redundant traffic from GM. TSB and TSB+SUF show higher dynamic energy consumption than prefetchers like IP-Stride or Berti because they trigger a greater number of prefetch requests, but they also achieve better performance.

6.2.4 Multi-core performance

Fig. 6.8 shows the performance of SUF on a 4-core simulated system. We plot the speedup of on-access Berti in a non-secure system (light gray), a secure cache system without prefetcher (red), on-commit Berti with SUF in a secure cache system (dashed dark light), and TSB+SUF (dashed black). The mixes have been sorted in increasing order of speedup. GhostMinion decreases the performance of the mixes by 16.8%. Only 14 mixes show a performance improvement (with a maximum of 13.7%).

All the secure prefetchers improve the performance in all traces over a secure system

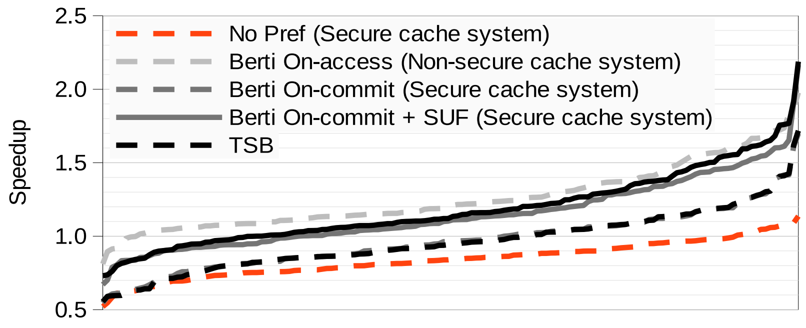


Figure 6.8: Speedup for 4-core mixes normalized to the non-secure cache system with no prefetching.

without prefetchers. As multi-core execution increases the traffic in the higher levels of the cache, it increases the latency of memory requests. Hence, the benefits of SUF reducing the traffic are more acute than in the single-core execution. TSB+SUF improves the performance over the baseline by 16.1%, followed by on-commit Berti+SUF (12.4%). The SUF filter improves performance over a secure cache system in all mixes and improves performance over the baseline in 78.0% of the mixes.

Conclusion

Secure cache systems play a crucial role in mitigating transient execution attacks such as Spectre by enforcing strict ordering. Despite the attention given to secure cache systems in recent years, hardware prefetching has often been treated as a secondary consideration. In this work, we argue that prefetching is essential for mitigating the negative impact of a secure cache system, and we conduct a comprehensive evaluation of state-of-the-art prefetch mechanisms in the context of a secure cache system.

Our analysis reveals two key findings: (i) existing secure memory hierarchies limit the potential of prefetchers, sometimes nullifying significant speedups, and (ii) prefetching techniques exhibit suboptimal performance when transitioning to the commit stage due to a loss of timeliness. To address these issues, we propose enhancements that significantly improve the effectiveness of hardware prefetchers. Our approach yields a 6.3% improvement in single-thread performance and a 23.0% enhancement in multi-core performance, incurring a modest hardware overhead of 0.59 KB per core.

Acknowledgements

Firstly, I express my gratitude to my advisor, **Prof. Biswabandan Panda**, for his support and invaluable guidance throughout my journey in this institution. His expertise, patience, and encouragement have been instrumental in shaping this research project and my academic growth.

I extend my appreciation to **Prof. Alberto Ros** and **Agustin Navarro-Torres** for their collaboration on this work. Their contributions, expertise, and insightful suggestions have immensely enriched the research and added depth to our findings.

I would also like to express my gratitude to my fellow **CASPER** friends, especially **Shubham Roy**. Our discussions, brainstorming sessions, and his valuable insights have played a crucial role in shaping the direction of this research.

Sumon Nath
IIT Bombay
10 December 2023

References

[1] "The 2nd data prefetching championship (dpc-2)," Jun. 2015. [Online]. Available: https://comparch-conf.gatech.edu/dpc2/

[2] "Micron dram power calculator," https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf, Dec. 2015.

[3] "SunnyCove microarhcitecture," https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove, May 2018.

[4] "SunnyCove microarhcitecture latency," https://www.7-cpu.com/cpu/Ice_Lake.html, May 2018.

[5] "The 3rd data prefetching championship (dpc-3)," Jun. 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/

[6] "SPEC CPU 2017 traces for champsim," https://hpca23.cse.tamu.edu/champsim-traces/speccpu/index.html, Feb. 2019.

[7] "ChampSim simulator," http://github.com/ChampSim/ChampSim, May 2020.

[8] "GAP traces for champsim," https://utexas.app.box.com/s/2k54kp8zvrqdfaa8cdhfquvcxwh7yn85/folder/132804598561, Mar. 2021.

[9] P. Aimoniotis, C. Sakalis, M. Själander, and S. Kaxiras, "Reorder buffer contention: A forward speculative interference attack for speculation invariant instructions," pp. 162–165, 2021.

[10] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *47th Int'l Symp. on Computer Architecture (ISCA)*, 2020, p. 132–144.

[11] S. Ainsworth, "Ghostminion: A strictness-ordered cache system for spectre mitigation," in *54th Int'l Symp. on Microarchitecture (MICRO)*, 2021, p. 592–606.

[12] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, 2009.

[13] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[14] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, Aug. 2015.

[15] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *26th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2021, p. 1046–1060.

[16] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13.

[17] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, Aug. 2018, pp. 991–1008.

[18] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019, pp. 249–266.

[19] Y. Chen, L. Pei, and T. E. Carlson, "Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *28th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2023, p. 16–32.

[20] P. Cronin and C. Yang, "A fetching tale: Covert communication with the hardware prefetcher," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 101–110.

[21] DDR, "Ddr standards," https://en.wikipedia.org/wiki/Double_data_rate. [Online]. Available: https://en.wikipedia.org/wiki/Double_data_rate

[22] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, “Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks,” *ACM Trans. Archit. Code Optim.*, 2012.

[23] J. Doweck, “Inside intel core microarchitecture and smart memory access,” in *Intel whitepaper*, 2006, pp. 1–13.

[24] A. Fog, “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers,” 2020, Available at https://www.agner.org/optimize/microarchitecture.pdf.

[25] B. Grayson, J. Rupley, G. D. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, “Evolution of the samsung exynos cpu microarchitecture,” in *47th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 40–51.

[26] D. A. Jiménez and C. Lin, “Dynamic branch prediction with perceptrons,” in *7th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 197–206.

[27] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Safespec: Banishing the spectre of a meltdown with leakage-free speculation,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[28] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” *CoRR*, vol. abs/1801.01203, Jan. 2018.

[29] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, “Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques,” in *2011 Int’l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

[30] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibánez, V. Viñals-Yúfera, and A. Ros, “Berti: an Accurate Local-Delta Data Prefetcher,” in *55thInt’l Symp. on Microarchitecture (MICRO)*, 2022, pp. 975–991.

[31] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *47th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 118–131.

[32] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “Pacman: Attacking arm pointer authentication with speculative execution,” in *49th Int’l Symp. on Computer Architecture (ISCA)*, 2022, p. 685–698.

[33] G. Reinman, B. Calder, and T. Austin, “Fetch directed instruction prefetching,” in *32nd Int’l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[34] G. Saileshwar, S. Kariyappa, and M. Qureshi, “Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning,” in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021, pp. 37–49.

[35] G. Saileshwar and M. K. Qureshi, “Cleanupspec: An "undo" approach to safe speculation,” in *52th Int’l Symp. on Microarchitecture (MICRO)*, 2019, p. 73–86.

[36] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander, “Ghost loads: What is the cost of invisible speculation?” in *16th Int’l Conf. on Computing Frontiers (CF)*, 2019, p. 153–163.

[37] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Efficient invisible speculative execution through selective delay and value prediction,” in *46th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 723–735.

[38] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Multi-lookahead offset prefetching,” in *The 3rd Data Prefetching Championship*, Jun. 2019.

[39] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *10th Int’l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[40] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, “Unveiling hardware-based data prefetcher, a hidden source of information leakage,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, p. 131–145.

[41] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers,” in *13th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 63–74.

[42] Standard Performance Evaluation Corporation, “SPEC CPU2017,” 2017. [Online]. Available: http://www.spec.org/cpu2017

[43] 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 *IEEE Symposium on Security and Privacy (SP)*, 2022.

[44] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, “Nda: Preventing speculative execution attacks at their source,” in *52th Int’l Symp. on Microarchitecture (MICRO)*, 2019, p. 572–586.

[45] J. Wikner, D. Trujillo, and K. Razavi, “Phantom: Exploiting Decoder-detectable Mispredictions,” in *56th Int’l Symp. on Microarchitecture (MICRO)*, 2023.

[46] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *51th Int’l Symp. on Microarchitecture (MICRO)*, 2018, pp. 428–441.