Secure Prefetching for Secure Cache Systems

Sumon Nath*, Agustín Navarro-Torres †, Alberto Ros†, Biswabandan Panda*

*Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India
†Computer Engineering Department, University of Murcia, Murcia, Spain
Email: sumon@cse.iitb.ac.in, agustin.navarro@um.es, aros@ditec.um.es, biswa@cse.iitb.ac.in

***Abstract*—Transient execution attacks like Spectre and its variants can cause information leakage through a cache hierarchy. There are two classes of techniques that mitigate speculative execution attacks: delay-based and invisible speculation. Invisible speculation-based techniques like GhostMinion are the high-performing yet secure techniques that mitigate all kinds of speculative execution attacks. Similar to a cache system, hardware prefetchers can also cause speculative information leakage. To mitigate it, GhostMinion advocates on-commit prefetching on top of strictness ordering in the cache system. Our experiments show that the GhostMinion cache system interacts negatively with the hardware prefetchers leading to redundant traffic between different levels of cache. This traffic causes contention and increases the miss latency leading to performance loss. Next, we observe that on-commit prefetching enforced by GhostMinion leads to performance loss as it affects the prefetcher timeliness.**

We perform the first thorough analysis of the interaction between state-of-the-art prefetching techniques and the secure cache system. Based on this, we propose two microarchitectural solutions that ensure high performance while designing secure prefetchers on top of secure cache system. The first solution detects and filters redundant traffic when updating the cache hierarchy non-speculatively. The second solution ensures the timeliness of the prefetcher to compensate for the delayed triggering of prefetch requests at commit, resulting in a secure yet high-performing prefetcher. Overall, our enhancements are secure and provide synergistic interactions between hardware prefetchers and a secure cache system. Our experiments show that our filter consistently improves the performance of secure cache systems like GhostMinion in the presence of state-of-the-art prefetchers (by 1.9% for single-core and 19.0% for multi-core for the top-performing prefetcher). We see a synergistic behavior of the filter with our proposed secure prefetcher, which leads to a further increase in performance by 6.3% and 23.0% (over the top-performing prefetcher), for single-core and multi-core systems, respectively. Our enhancements are extremely lightweight incurring a storage overhead of 0.59 KB per core.

I. Introduction

Transient execution attacks, pioneered by Spectre [28] but followed rapidly by other attacks [9], [15], [17], [18], [33] take advantage of the cache state affected by *transient* instructions. Transient instructions are speculative instructions that do not commit. Speculative execution is a fundamental technique used by high-performance processors, and hence, it cannot be disabled in pursuit of security. To mitigate the speculative execution attacks that exploit the cache, various proposals [10], [11], [27], [36]–[38], [45], [46], [48] strive to provide security with minimal performance loss. In general, there are two kinds of mitigation techniques proposed in the literature: delay-based and invisible speculation. In delay-based approaches, the *transmission* of secret-dependent values is stalled until it is considered *safe* to proceed. The determination of safety can be complex and requires sophisticated mechanisms to accurately identify when an instruction can be considered safe for execution. In invisible speculation, secret-dependent loads are permitted to execute. However, the effects of these executions are concealed from the cache hierarchy and other microarchitectural structures.

Among all the proposals, GhostMinion [11], Speculative Taint Tracking (STT) [48], and Non-speculative Data Access (NDA) [45] are the strictest as they mitigate backward-in-time attacks such as speculative interference attacks [15]. Between STT, NDA, and GhostMinion, GhostMinion is the lightweight and high-performing mitigation technique. GhostMinion is an invisible speculation technique that enforces a strictness ordering that ensures the mitigation of varieties of speculative execution attacks through the cache system: cache hierarchy, miss status holding registers (MSHRs), and hardware prefetchers [10]. GhostMinion uses a small speculative cache (GM) that stores the data corresponding to speculative loads, and when a load commits, the data is communicated to L1D. When the same data is evicted from L1D, the data is communicated to the L2, and on eviction from L2, the data is communicated to the LLC. On average, GhostMinion incurs a performance loss of around 5% as compared to a non-secure cache system.

Data prefetchers play an important role in improving cache performance by converting cache misses into hits. Recent advances in data prefetchers have pushed the limit of single-thread performance with average performance boosts of 3% to 5% [13], [16], [31], [32]. In the last decade, two ISCA championships on data prefetching [1], [5] have helped in this trend. Unfortunately, hardware prefetchers, which are trained and triggered on speculative loads, can also be used as a source of information leakage even on a secure cache system [10], [11]. A speculative attack using prefetchers works as follows: (i) The attacker primes the cache; The corresponding cache has a prefetcher; (ii) The victim loads secret data similar to the Spectre attack; (iii) The speculative load generated by the victim trains and triggers the hardware prefetcher; (iv) The prefetcher request data as per its address prediction that comes to the cache; (v) Finally, the attacker probes the cache.

GhostMinion makes a case for secure hardware prefetching through *on-commit* prefetching: A secure prefetcher should be trained on commit and prefetching should only happen on commit. This way the prefetcher will not affect the cache and MSHR state speculatively, and transient instructions cannot exploit the prefetcher for information leakage. We show that data prefetching can indeed alleviate the performance loss of a secure cache system. Despite the importance of data prefetching, no detailed study has been carried out about the impact of secure prefetching techniques.

In this paper, we analyze for the first time the interaction between a wide range of state-of-the-art hardware prefetchers and a high-performing secure cache system. We evaluate IP-stride [12], the well-known prefetcher used in industry, Bingo [13], SPP+PPF [16], IPCP [32] (winner of the 3rd data prefetching championship [5]), and Berti [31], on a secure cache system like GhostMinion. Berti is the state-of-the-art L1D prefetcher (with an accuracy of almost 90%) that orchestrates its requests across the cache hierarchy. We discover that prefetchers interact negatively with secure cache systems like GhostMinion. We find that two main factors prevent them from reaching their optimal performance and propose microarchitectural solutions to overcome them.

Our observations. First, we analyze the performance improvements of the evaluated prefetchers both on a non-secure cache system and a secure cache system like GhostMinion for both SPEC CPU 2017 and GAP workloads (see Section VI for simulation details). We observe that prefetching techniques improve performance both for secure and non-secure cache systems, but the gap between them is high. The performance gap is because of an increase in memory access latency due to additional memory traffic introduced to update the cache hierarchy with invisible loads. On average, GhostMinion introduces additional traffic of more than 1.5× to L1D when compared to a non-secure cache system with hardware prefetchers (Section III).

Next, we analyze the impact of implementing a secure prefetcher on a secure cache system, that is, the impact of training and prefetching on-commit, instead of on-access. Fig. 1 shows the performance improvements obtained by our prefetchers in a secure cache system when they are trained and triggered on-access (second bar) and on-commit (third bar). We observe a consistent performance loss of 3%-4% for all prefetchers with training/prefetching on-commit compared to prefetching on-access. We find that the key factor is timeliness, not the inability to capture the applications' access patterns (Section III). A major part of the performance loss for on-commit prefetching is due to a new class of late prefetch requests which we coin as "commit-late": misses whose prefetching had not been initiated when the processor requested the data, but that would have been initiated if the prefetch request had been triggered on access. In summary, on average, compared to on-access prefetching on a non-secure cache system, we see a performance loss of around 10% with

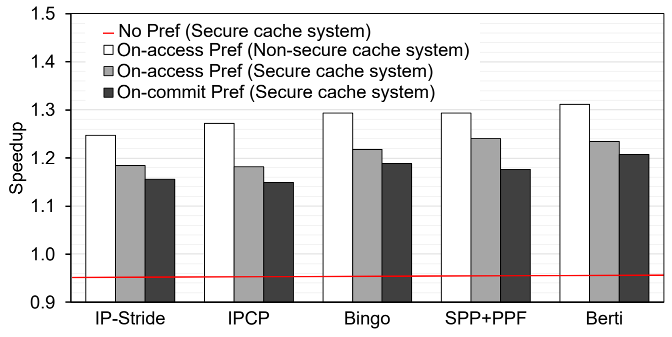


Fig. 1. Speedup of state-of-the-art prefetchers normalized to a non-secure cache system with no prefetching.

on-commit prefetching on a secure cache system.

Our contributions. In this paper, we shed some light on the reasons behind low-performance secure prefetchers and propose a low-cost yet effective solution to recover the performance loss and enable the full potential of secure prefetching, closing the gap to non-secure cache systems.

This paper brings the following contributions:

- We show that prefetchers lose relative performance on a secure cache system due to (i) the additional memory traffic introduced by the secure cache system and (ii) prefetch timeliness issues (Section III).
- We propose a mechanism to filter out the superfluous non-speculative updates performed in a secure cache system. Our filter is lightweight and incurs a storage overhead of 0.12KB (Section IV).
- We propose a mechanism to ensure the timeliness of a prefetcher that is trained and issues prefetch requests at commit, making a case for a secure yet timely and high-performing prefetcher. The end result is the first high-performance secure prefetcher with a storage overhead of 0.47KB (Section V).
- We show that our enhancements on top of the state-of-the-art prefetcher helps in bridging the performance gap between a non-secure cache system and a secure cache system. For SPEC CPU2017 and GAP benchmarks, our enhancements improve performance by 6.3% (our filter contributes to around 30% of the improvement and the rest comes from the better-trained on-commit prefetcher). For a 4-core system, our mechanisms improve performance by 23.0% over on-commit state-of-the art prefetcher in a secure cache system (Section VII).

II. Background & Related Work

A. Threat model

We assume the following capabilities in our *transient execution* attacker: (i) (S)he is capable of mounting attacks like Spectre and speculative interference [15] through the cache system. (ii) (S)he can exploit a hardware prefetcher by speculative training and prefetching that can change the cache state, as mentioned in Muontrap [10]. (iii) The attacker and the victim can be part of the same process or two different

TABLE I

SUMMARY OF MITIGATION TECHNIQUES. *WE USE THE SECURE IMPLEMENTATION OF GHOSTMINION AS SUGGESTED IN PENSIEVE [47]. DOM, MUONTRAP, INVISISPEC, AND CLEANUPSPEC ARE NOT SECURE AS THESE TECHNIQUES DO NOT MITIGATE THE SPECULATIVE INTERFERENCE [15] ATTACKS. WE CATEGORIZE PERFORMANCE SLOWDOWN INTO THREE BINS: LOW (<5%), MEDIUM (5% TO 10%), AND HIGH (>10%).

Mitigation techniques	Classification	Secure?	Storage overhead	Performance slowdown
CleanupSpec [36]	Undo-based	No	<1KB	Medium
NDA [45]	Delay-based	Yes	≈ 150 bytes	High
STT [48]	Delay-based	Yes	≈ 1.4 KB	Medium
NDA + Doppelganger [29]	Delay-based	Yes	≈ 13.5 KB	Medium
DoM [38]	Delay+invisible speculation-based	No	≈ 0.4 KB	High
DoM + Doppelganger [29]	Delay+invisible speculation-based	No	≈ 13.9 KB	High
STT + Doppelganger [29]	**Delay-based**	**Yes**	**≈ 14.9 KB**	**Low**
InvisiSpec [46]	Invisible speculation	No	≈ 9.5 KB	High
MuonTrap [10]	Invisible speculation	No	2 KB	Low
GhostMinion* [11]	**Invisible speculation**	**Yes**	**2 KB**	**Low**

processes. The attacker can run arbitrary code but cannot access secret data directly, i.e., the attacker is running within a sandbox either at the user or at the kernel level. (iv) There are timing-based side and covert channels involving hardware data prefetchers and caches [19], [20], [41] that can be mitigated by existing spatial isolation techniques [22], [35].

B. Recent mitigation techniques

A self-contained Table I summarizes recent mitigation techniques keeping security, performance, and storage in mind. As mentioned in Section I, mitigation techniques fall into one of the two broad approaches: delay-based [38], [45], [48] and invisible speculation [10], [11], [37], [46]. For performance evaluation, we use SPEC CPU2017 [43] and GAP [8] benchmarks. Among all the delay based approaches, STT provides security guarantees with minimum performance overhead. STT operates on the premise that it is safe to forward the *secret* data to dependent speculative loads unless those instructions are forming a covert channel. In the case of explicit covert channels, it simply blocks the LOADs whose source registers contain speculatively derived value. A recent performance enhancement technique called Doppelganger [29] improves the performance of delay-based approaches, which includes STT. Compared to delay-based techniques that incur high storage overhead, invisible speculation techniques like GhostMinion entail lower performance penalties with minimum storage overhead. Therefore, in this paper, we select GhostMinion as our secure cache system.

C. The GhostMinion secure cache system

GhostMinion uses a small 2KB cache called the GM, accessed concurrently along with the L1D, that stores the data of speculative instructions till they commit (or retire). On a demand miss generated by a speculative instruction at GM, it searches for the data at L1D, L2, and LLC similar to a conventional cache hierarchy. However, on a hit at L1D, L2, or LLC, the cache state (replacement policy priority bits) is not updated. On a miss at the L1D, L2, and LLC, the response is directly filled into GM, bypassing L1D, L2, and LLC (Fig. 2, ❶). On a commit, the data of the committed instruction is transferred to the L1D cache from GM if it is a GM hit, through *on-commit* writes. If the data is later evicted

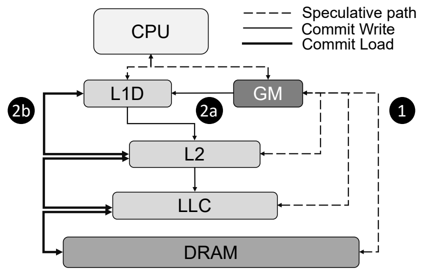


Fig. 2. GhostMinion secure cache system

from L1D it is then moved to the L2 cache, with further evictions at the L2 leading to its communication to the LLC (Fig. 2, 2a). In case of a GM miss, *re-fetching* of data is done into the non-speculative cache hierarchy (L1 to LLC) (Fig. 2, 2b). GM is neither inclusive nor exclusive to the rest of the cache hierarchy. Within GM, instructions are restricted to see the eviction or insertion of others depending on their temporal order. The temporal order is maintained based on the timestamp. TimeGuarding used by GhostMinion makes sure the insertions and the evictions are invisible under multiple speculations. To hide contention at MSHRs, the timestamp metadata propagates into the MSHRs at each cache level, allowing younger loads to be canceled and replaced by the older loads (leapfrogging). Also, in GhostMinion, a block can only be in a shared or invalid state and the coherence states of GM and non-speculative caches are not altered until an instruction is committed.

III. MOTIVATION

A. Impact of secure cache system on hardware prefetching

This section analyzes what prevents the prefetcher effectiveness on secure cache systems like GhostMinion. Fig. 3 shows the increase in L1D accesses for the prefetchers evaluated in this work with a GhostMinion secure cache system and for on-access prefetching. On a non-secure system with no prefetching, the average L1D accesses per kilo instructions (APKI) is 199, which goes up to 375 in GhostMinion because of commit requests that update the cache state as discussed in Section II-C. With Berti on GhostMinion, the APKI goes up to 570. The trend persists for all the prefetchers. For L2 prefetchers like Bingo and SPP+PPF there is no access from the prefetcher to L1D as the prefetch requests are generated from L2.

The increase in APKI results in additional traffic causing an increase in L1D miss latency as shown in Fig. 4. One of the primary contributors to this additional miss latency is the following interesting trend that makes the latency worse especially in the presence of hardware prefetching. On average, for the Berti prefetcher, with a secure cache system, there is a 10.4% increase in L1D MSHR occupancy and the L1D MSHR becomes full for an additional 8.7% of the time. Also, without prefetching, the L1D MSHR occupancy decreases by 15.9% when we move from a non-secure to a secure system because demand misses are first served by the GM.

To dig deep into the interesting interactions, we choose `605.mcf_s-1554B` and perform a detailed analysis. Fig. 5(a) shows the normalized performance with respect to a non-secure baseline without prefetching. A significant reduction in performance is observed when the prefetchers are applied to a secure cache system (for Berti, it is more than 300%). Fig. 5(b) shows the increase in traffic at L1D contributed by load, prefetch, and commit requests from GhostMinion. Fig. 5(c) shows a significant increase in L1D miss latency. When we analyze the MSHR occupancy numbers, without prefetching, L1D MSHR occupancy decreases by 16.2% when we move from a non-secure to a secure cache system because the demand requests are first served by GM. However, with prefetching, there is an increase in L1D MSHR occupancy of 10.1% when we move from a non-secure to a secure cache system. This happens because, with a non-secure cache system, the L1D MSHR only has to deal with demand and prefetch requests, while with a secure cache system, it also has to handle prefetch requests on top of GhostMinion requests, which increases the pressure on the MSHR. Without prefetching, L1D MSHR is almost never full. However, with prefetching, there is an increase in the percentage of time L1D MSHR is full (from 6.3% to 20%). In Section IV, we propose a mechanism that resolves the additional traffic-induced performance loss when hardware prefetching is enabled.

B. *Impact of secure hardware prefetching*

As described in GhostMinion [11], on-commit prefetching results in no leaking of information due to speculative execution. However, as shown in Fig. 1, moving state-of-the-art prefetchers to the commit stage results in 3%-4% performance loss compared to on-access prefetching.

Fig. 6 shows the average demand misses per kilo instructions (MPKI) across the workloads analyzed in this work. The MPKI is shown for the cache level where the prefetcher works, namely, L1D for IP-stride, IPCP, and Berti, and L2C for Bingo and SPP+PPF. In addition, each prefetcher is evaluated both with on-access and on-commit prefetching. The MPKI has been divided into the following four categories:

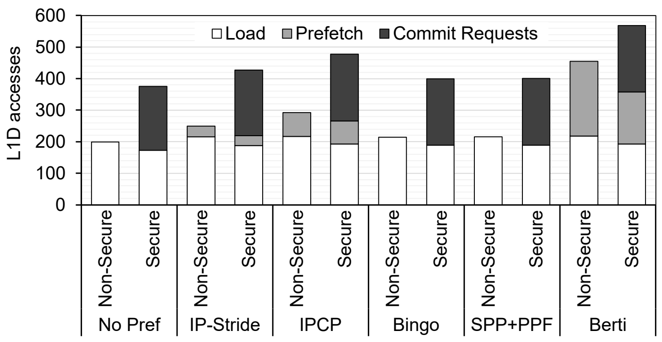


Fig. 3. Average L1D accesses per kilo instructions with on-access prefetching

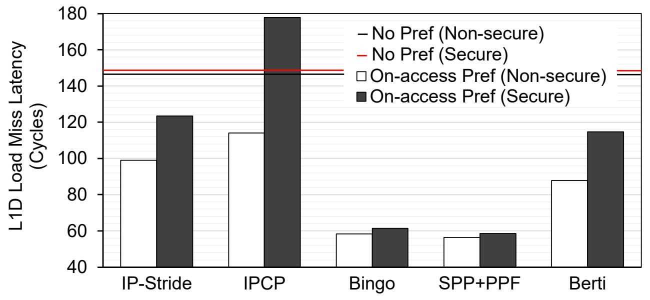
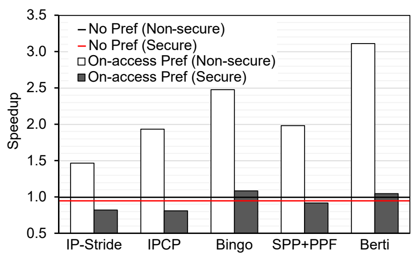


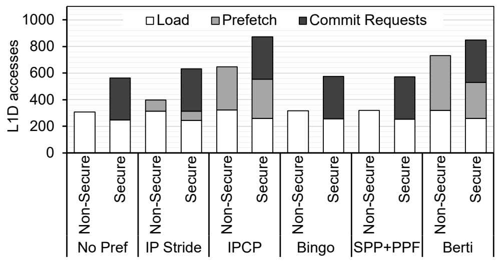
Fig. 4. Average L1D load miss latency with on-access prefetching

(i) *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.
(ii) *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.
(iii)*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 of training at commit.
(iv) *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 (excluding the MPKI coming from the commit late prefetch requests) is lower for on-commit prefetchers. The Uncovered misses of L1D/L2-MPKI are reduced with on-commit as some uncovered ones are now classified as on-commit late.

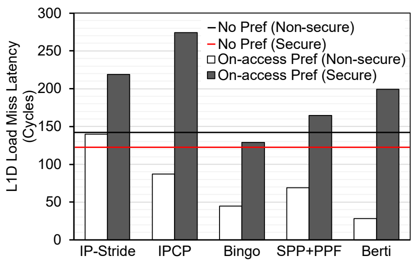
Despite this trend, performance is worse when compared to on-access prefetching. The reason is timeliness. Although traditional late prefetch requests practically do not increase



(a) Speedup



(b) L1D traffic



(c) L1D Load miss latency

Fig. 5. 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

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. Section V proposes a mechanism that mitigates the lack of timeliness.

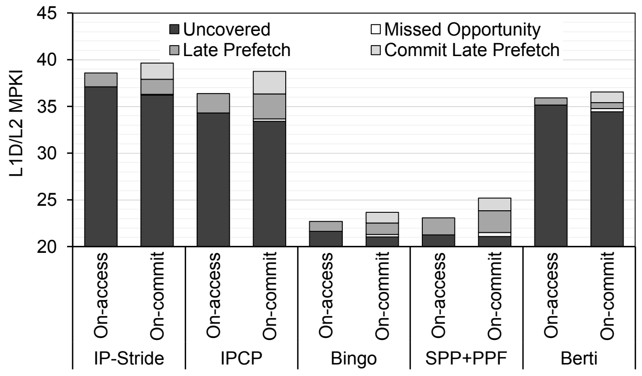


Fig. 6. Average L1D/L2 demand MPKI in terms of coverage and lateness.

IV. Prefetch-friendly secure cache system

Secure cache systems, based on invisible speculation, update the cache hierarchy when memory instructions are not speculative, e.g., when committing. In the case of GhostMinion, this entails re-fetching the cache line on a miss in the GM or sending on-commit write requests (for clean cache lines) to the rest of the cache hierarchy on a hit in the GM as discussed in Section II-C. The goal of this extra data movement is to populate the cache hierarchy, which was left intact when the data was speculatively requested by the core, to minimize cache misses in the subsequent accesses.

Both re-fetching and write propagation have an important impact on memory hierarchy traffic. The extra traffic, however, does not come with a noticeable performance degradation in a memory system that is not heavily contended, as one with prefetching mechanisms. However, prefetching mechanisms stress the cache hierarchy queues and MSHRs, preventing the prefetcher from improving performance as shown in Section III-A. We observe that many requests aimed at restoring the cache hierarchy are indeed not necessary and cause severe contention. For example, triggering a re-fetch for data that was provided by the L1D would consume L1D ports to just update the LRU replacement policy. In the same context, the on-commit write requests propagate up in the memory hierarchy until the data is already found in a cache level. The access to the cache level already containing the cache line could be therefore avoided. Driven by this observation, we propose the secure update filter (SUF). SUF records the cache level that provided the data when requested. Then, at commit time, either filters the re-fetching when the data was provided by the L1D or stops the on-commit write propagation at the level previous to the one that provided the cache line. In case the SUF mispredicts, because the fetched cache line may have been evicted in the interim, a subsequent fetch request would incur extra latency since it will be served from a higher level. Thanks to SUF's high accuracy, the number of cache accesses is reduced, and consequently, the amount of traffic generated. SUF works independently of the underlying prefetching technique, in a transparent way.

Identifying the cache level holding a cache line. SUF uses the lower level (L1D is the lowest level and LLC is the highest level of the cache) holding a cache line to decide if filtering should be employed. The cache level can be learned when the processor requests the data, by propagating down the hierarchy the cache level that served the cache line. That

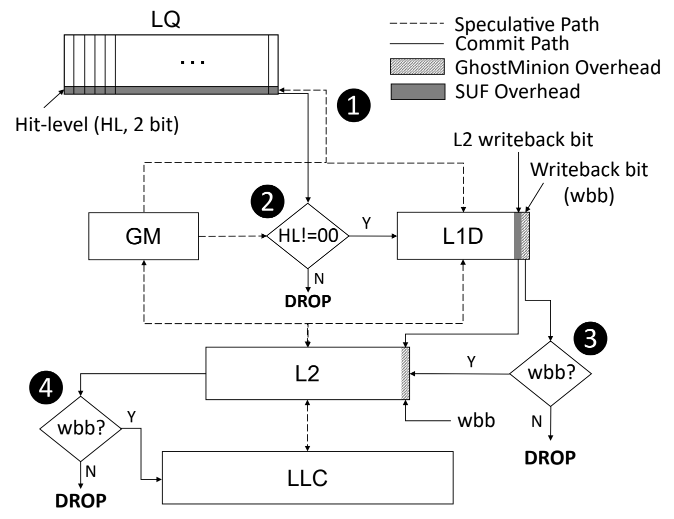


Fig. 7. Overview of the secure update filter (SUF)

information is encoded using 2 bits indicating if the data comes from L1D (or GM, which is accessed in parallel), L2C, LLC, or DRAM. The 2-bit hit-level information is stored along with the requested data in the memory operation entry at the load queue (LQ) (Fig. 7, step ❶).

Filtering updates. Once a speculative load is committed, it checks the GM in order to decide if re-fetching the cache line down the hierarchy (GM miss) or propagating the cache line up in the hierarchy (GM hit) is required. SUF checks the hit-level field, and proceeds as follows. In the case of the data being provided by the L1D (value 00), SUF drops the update (both for re-fetching and on-commit propagation). Otherwise, the re-fetch or propagation is done as usual that causes the cache line to move from GM to L1D (step ❷). Upon eviction of the cache line from either L1D or L2, the decision to propagate the writeback block is determined by the GhostMinion *writeback bit*. The value of the GhostMinion writeback bit at L1D and L2 are evaluated at commit time using the hit-level and propagated along with the writeback block. Each cache line at L1D stores the L2 writeback bit as well, so that it gets propagated to L2 upon writeback (step ❸). Finally, during the eviction from L2, the GhostMinion writeback bit is once again employed to determine whether to propagate or not (step ❹).

Applicability. SUF is applicable to any secure cache system based on invisible speculation that updates the cache hierarchy on commit.

Storage overhead. SUF is implemented with only 0.12 KB of additional storage: 0.03 KB at the LQ and 0.09 KB at the L1D. Each of the 128 entries in the LQ is extended with a two-bit hit-level field and each of the 768 entries of the L1D is extended with a single L2 writeback bit.

V. Timely secure prefetcher

As discussed in Section III-B, transitioning prefetchers from on-access to on-commit leads to an average performance loss of 3% to 4%. This performance degradation stems from the new timeliness constraints, which introduces *commit-late* prefetch requests and missed prefetching opportunities. These factors contribute to lower prefetch accuracy and coverage. We observe that this behavior can be fixed by adjusting prefetch timeliness for prefetchers like IP-stride and IPCP. One of the ways to improve prefetch timeliness is to increase the prefetch distance to cover more distant memory requests, which can lead to fewer commit-late prefetch requests. However, we find that even with adaptive approaches, all the prefetchers fail to beat the Berti prefetcher.

Another option for addressing the timeliness issue is to modify the learning process of the hardware prefetcher. We pursue this approach with Berti. We focus on Berti because (i) it shows the best performance improvements over its peers and (ii) its learning is unaffected by the order of memory access streams (on-access vs on-commit) as it operates on timely deltas, which is not the case with other prefetchers.

A. Berti prefetcher: 10K feet view

Training the prefetcher. Berti trains for deltas for a given IP, which is known as the local deltas. For an IP, local deltas are defined as the difference between the cache line addresses of two demand accesses. The goal of the training mechanism is to estimate the coverage of each delta per IP, considering those deltas that would result in a timely prefetch. The training consists of measuring fetch latency, learning timely deltas, and computing the coverage of the deltas.

1. Measuring fetch latency. In order to learn the deltas that are *timely*, it is necessary to measure the time required to fetch data to the L1D. This measurement is performed for any cache line in L1D, both for demand misses and prefetch requests. Fetch latency can be measured by keeping a timestamp for any L1D miss inserted into the MSHR and prefetch request inserted into the PQ. On an L1D fill, the latency is computed by subtracting the stored timestamp from the current cycle.

2. Learning timely and accurate deltas. Once the fetch latency is obtained for each L1D fill, Berti looks up past accesses (from the same IP) that could have triggered a timely prefetch for the current request. Timely deltas are then computed by subtracting the address of each timely access in the history from the current address.

3. Computing the coverage of deltas. On every search in history, Berti obtains a set of timely deltas. Deltas that frequently appear in the search would cover a significant fraction of misses, while deltas that rarely appear would result in low coverage. It is important to note that high local (per IP) coverage translates into high global accuracy.

Issuing prefetch requests. For a given IP, deltas with highest coverage are selected, added to the current load address to form the prefetch requests. Berti orchestrates the prefetch requests across the cache hierarchy depending on the coverage of each delta and the L1D MSHR occupancy.

B. Issues with the secure prefetcher

As explained in the previous subsection, Berti relies on the fetch latency to guide its learning mechanism. For secure

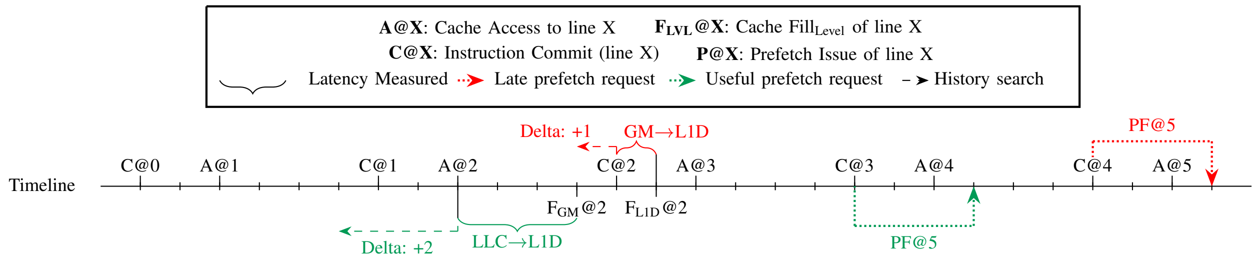


Fig. 8. Timeline representing working of on-commit Berti (red) and TSB (green). The timeline represents the access/commit of consecutive executions of the same load instruction, with each step representing one cycle.

prefetching, we identify two main issues: (i) the latency seen by the prefetcher is a misleading on-commit latency (instead of the actual on-access latency) and (ii) the deltas selected are timely at commit but not at access when data are needed.

To understand these problems better we start by describing the on-commit version of Berti implemented on GhostMinion. Berti is located at the L1D cache and utilizes post-commit L1D accesses and fills for training. Since GhostMinion speculatively fills the GM and moves that cache line to L1D on commit, Berti observes the on-commit write latency from the GM to L1D, instead of the fetch latency from a higher level of the memory hierarchy to GM. This alteration in fetch latency disrupts the learning process, leading to inaccurate delta learning and prefetch requests, which in turn translates into performance degradation. Fig. 8 (in red) provides a visual example of the training and prefetch request issuance process for on-commit Berti in GhostMinion. The timeline shows a +1 delta for a given load. All L1D accesses are misses that take three cycles to fill GM and one cycle for on-commit write from GM to L1D. We focus on the process of how on-commit Berti issues a prefetch request for cache line @5.

Upon the commit of the load for @2 (C@2), the learning process initiates. When cache line @2 fills the L1D cache from the GM (via on-commit write), the latency of one cycle is recorded. Consequently, Berti searches for the nearest commit instruction capable of triggering a timely prefetch request. In this example, this instruction is C@1. Berti then learns the appropriate delta, for the writeback latency, which in this case is +1. From this point forward, 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 late prefetch request since the access to cache line @5 is performed two cycles later. This occurs because the learning process is performed (i) with the writeback latency, not with the fetch latency, and (ii) the deltas selected are timely at commit (C@2) but not at access (A@2). Note that this late prefetch will occur even if Berti searches for deltas based on the cache access time (A@2) rather than the commit time (C@2). Hence, both problems should be addressed in order to achieve timely secure prefetching.

C. *Timely training of the secure prefetcher*

Driven by the previous two observations, we propose Timely Secure Berti (TSB), a new timely training mechanism for Berti that utilizes the fetch latency to GM and computes the right delta using the access times. This way the training mechanism emulates the latency that future demand accesses will face.

TSB works as follows: first, when the demand load miss happens, it speculatively saves the necessary information for training Berti correctly, including the access time and the fetch latency to GM. At the time of commit, the commit time for each demand miss is saved in history table. While training at commit, TSB searches committed instructions, which could have triggered a timely prefetch for the current load. Timely deltas are calculated by subtracting the address of each timely commit from the current address. As we show in Fig. 8(in green), when cache line @2 fills the GM, the fill latency is calculated as the difference between the fill timestamp and the cache access timestamp and saved until commit. When instruction @2 commits (C@2) and Berti is ready to be trained: the fill latency is used to search the commit that can trigger a timely prefetch request that will hit in the access of cache line @2; once the commit is known (in our example, C@0), the delta is calculated (+2). Unlike the default on-commit version, TSB will trigger prefetch requests with delta +2, which means that C@3 triggers the prefetch request for cache line @5, resulting in a timely prefetch request.

Storage overhead. TSB uses X-LQ, an extension of the LQ needed to propagate the actual fetch latency. The *X-LQ* is dual-ported and it contains as many entries as the LQ (128 entries in our modeled system) and it is indexed with the LQ entry id (one-to-one mapping). Each entry contains a valid bit, a bit indicating that the access was a hit on a prefetched cache line (Hit_p), a 16-bit access timestamp, and a 12-bit fetch latency. TSB incurs a storage overhead of only 0.47 KB over Berti (3.01 KB over no-prefetch). Fig. 9 shows an overview of TSB, where the light gray represents the Berti hardware and the dark gray represents the additional TSB hardware.

On an L1D miss, the valid bit is set, and the access timestamp is filled with information from the clock of the local processor (the last 16 bits of the current cycle). When the cache fill in the GM is done, the fetch latency is also recorded. On a hit to a prefetched cache line, both the valid bit and the Hit_p bit are set. In this case, the access timestamp is

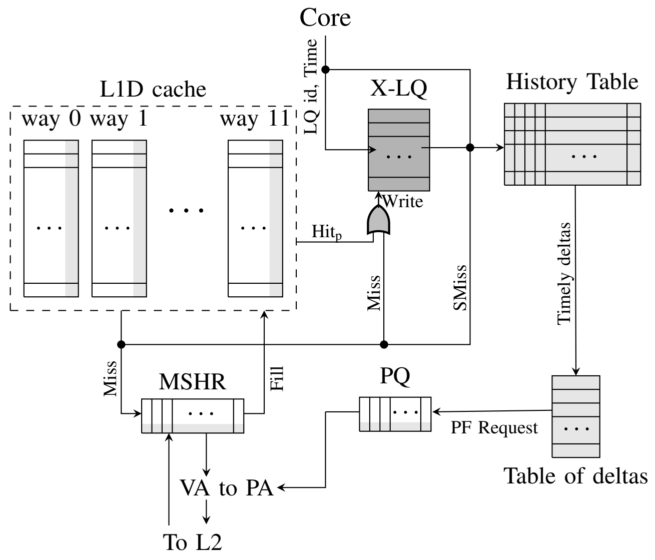


Fig. 9. TSB design overview. Berti original hardware extensions are shown in light gray. TSB hardware extensions are shown in gray.

TABLE II
SIMULATION PARAMETERS OF THE BASELINE SYSTEM.

Core	Out-of-order, hashed perceptron branch predictor [25], 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, write watermark: 7/8th, Chip: 4 KB row-buffer, open page, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

also recorded, but the fetch latency corresponds to the latency of the prefetched line (which has been previously computed and stored along with the L1D cache [31]). On a regular hit, the valid bit is not set, since no action will be taken at commit. When a load commits, the history table is filled for both misses and prefetch hits using commit timestamp, and the timely deltas are searched by using the fetch latency and access timestamp in X-LQ.

TSB security guarantees. TSB is trained and triggered on commit, which ensures that it is not trained on any transient instructions, and hence no prefetch requests can be generated based on transient information. In the speculative phase, the access time and fill latency of a particular request are stored in the X-LQ. The only vulnerable information is fill latency, which may be altered by a transient instruction. Any aberration to the fill latency through resource contention by a transient instruction could lead to prefetcher-based side channels. If the fill latency of a bound-to-commit instruction is altered by a transient instruction (e.g. backward-in-time attack), then the security is compromised. However, this is not possible with GhostMinion as a transient instruction can not affect the timing of a bound-to-commit instruction thanks to strictness ordering ensured by time-guarding. Also, as the X-LQ is flushed on a domain switch, the transient information stored in the X-LQ cannot be exploited by a malicious process. Moreover, the information stored by a particular load instruction in the X-LQ is accessible only by that instruction and only at commit time. No other instruction can access the information corresponding to any other instructions. This makes it secure as there is no possibility of data leakage.

TSB applicability. TSB applies to all secure cache systems (both invisible speculation and delay-based).

D. Timely training of non-self-timing prefetchers

Berti is a self-timing prefetcher. However, prefetchers like IP-stride, IPCP, Bingo, and SPP+PPF are not. To make these prefetchers secure, they should be trained and triggered on commit. The prefetch tables should not be updated on speculative requests. To compensate for the performance loss as shown in Figure 1, we make them timely.

IP-stride and IPCP. For prefetchers like IP-stride and IPCP, the prefetch distance should be tuned based on the lateness. We use a mechanism that increases the distance with an increase in prefetch lateness. We calculate prefetch lateness as the ratio of late prefetch requests to useful prefetch requests. We monitor prefetch lateness every 512 misses (size of the L1 in terms of the number of cache lines) and if the prefetch lateness increases for two consecutive intervals, then we increment the prefetch distance by one. Updating distance based on the lateness of only the previous interval leads to noisy decision-making.

SPP+PPF. For SPP+PPF, we use the same mechanism of prefetch lateness-driven adaptive distance selection as done with IP-stride and IPCP. However, SPP is a different prefetcher that uses the predicted delta in the signature used for finding out the next delta, recursively. To make it adaptive, we continue the learning of SPP with on-commit requests. However, we skip the next k deltas before we start prefetching, where k is driven by the prefetch lateness. Based on empirical analysis, we find that for timely prefetching, the value of k is between two to five. As SPP is an L2 prefetcher, the monitoring interval used is 4096 misses (one-half of the size of the L2).

Bingo. Bingo is a region-based prefetcher, similar to SMS [42] where introducing timeliness is a non-trivial task. We extend Bingo with temporal information as suggested in Tempo [44] using a local tempo buffer and global tempo buffer. We then change its distance dynamically based on the prefetch lateness.

For all the prefetchers, we use the lateness threshold of 0.14, which is just less than the average lateness while we perform on-commit prefetching. However, with Bingo, the prefetch lateness threshold that we use is 0.05. In general, with Bingo, the number of late prefetch requests is lower than IP-stride, IPCP, and SPP+PPF, as shown in Figure 6. We also use a phase change detector as used in prior works [26] and on an application phase change, we reset the prefetch distance to the base distance used.

TABLE III
CONFIGURATIONS OF EVALUATED PREFETCHERS

Prefetcher	Configuration	Size
IP-Stride	1024 entries	8KB
IPCP [32]	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 and reject tables	39.2 KB
Berti [31]	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

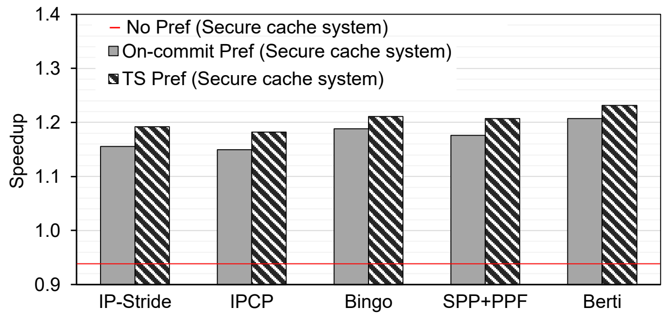


Fig. 10. Speedup of timely secure (TS) version of state-of-the-art prefetchers normalized to a non-secure cache system with no prefetching

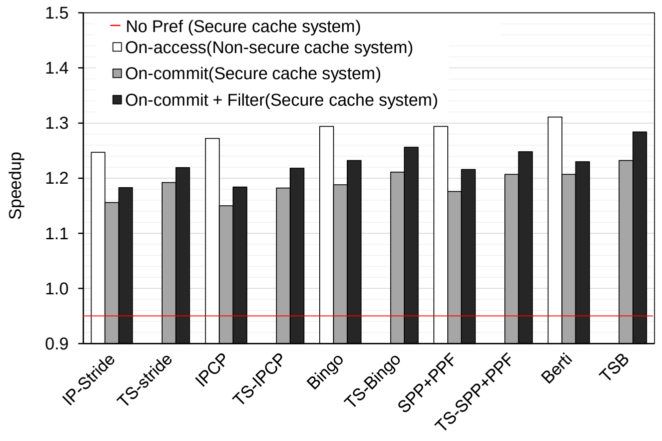


Fig. 11. Speedup normalized to non-secure cache system with no prefetching.

VI. METHODOLOGY

We use ChampSim [7], a trace-driven simulator used for the 2nd and 3rd Data Prefetching Championships (DPC-2 [1] and DPC-3 [5]). Recent prefetching proposals are also coded and evaluated on ChampSim [13], [16], [31], [32], [39]. The version employed in DPC-3 has been extended with a decoupled front-end [34], a detailed memory hierarchy support for address translation, and a faithful DRAM model. We calculate the dynamic energy consumption of the memory hierarchy (caches and DRAM) with CACTI-P [30] and the Micron DRAM [2] power calculator on 7 nm process technology. Table II details our baseline system configuration, similar to an Intel Sunny Cove microarchitecture [3], [4], [24].

We employ publicly available traces [6], [8] from the SPEC CPU2017 [43] and single-threaded GAP [14] benchmark suites. We limit our study to 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 LLC in our baseline system. We run both single- and multi-core simulations. We collect statistics for 200M sim-point instructions after a 50M-instruction warm-up [40]. For multi-core, we simulate 150 randomly generated heterogeneous mixes of SPEC CPU2017 and GAP traces and report weighted speedup.

We evaluate the effectiveness of SUF and TSB on a GhostMinion [11] secure cache system with a 2KB GM with 1 cycle latency for different data prefetchers: IP-Stride [23] (the Intel and AMD L1D prefetcher), IPCP [32], Bingo [13], SPP+PPF [16], and Berti [31]. We use the tuned implementations of each prefetcher using the parameters listed in Table III.

VII. RESULTS

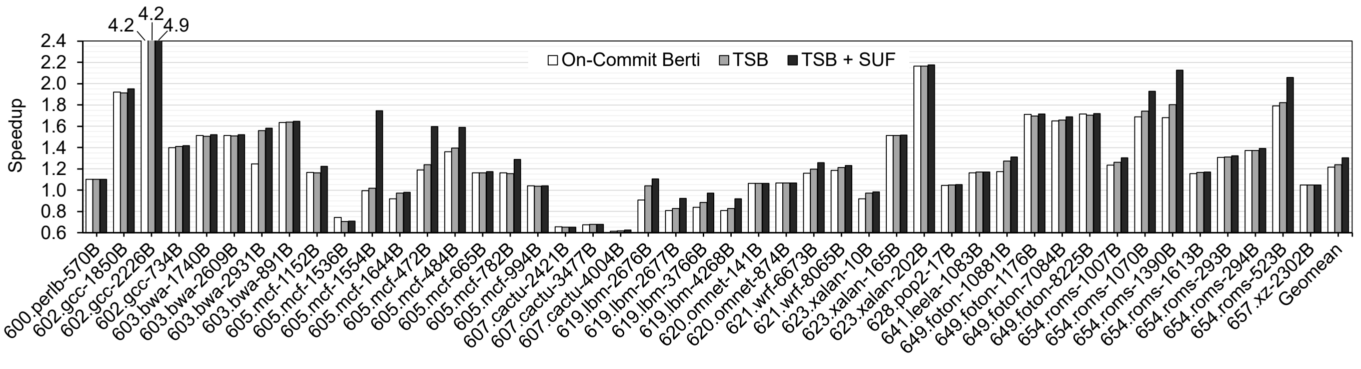
This section shows the benefits of our two main contributions: the secure update filter (SUF) and the timely secure Berti (TSB) prefetcher. All normalized graphs are relative to a non-secure system without prefetching. If a red line is present, it represents a GhostMinion secure cache system without 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).

A. *Performance*

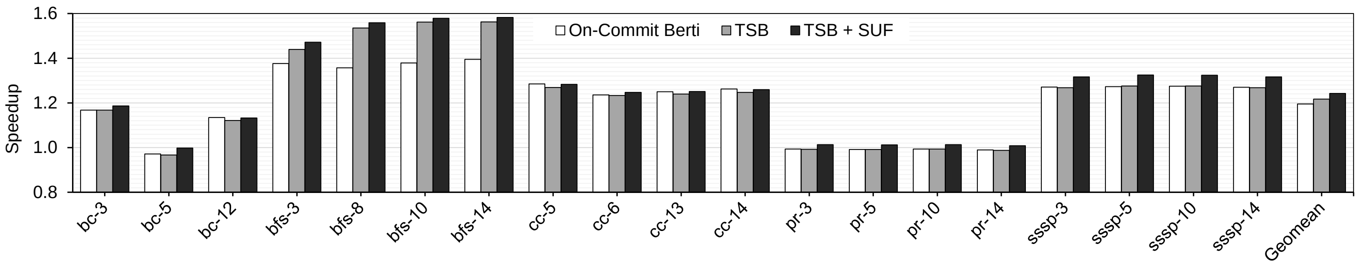
Speedup with timely secure prefetching The average single-thread speedup of the prefetchers with their timely secure versions is shown in Fig. 10. We name the timely secure versions of all the evaluated prefetchers as TS-stride, TS-IPCP, TS-Bingo, TS-SPP+PPF, and TSB. With GhostMinion as secure cache system, TS-stride, TS-IPCP, TS-Bingo, and TS-SPP+PPF outperform on-commit versions of IP-stride, IPCP, Bingo, and SPP+PPF by 3.1%, 2.84%, 1.92%, and 2.64%, respectively as shown in Fig. 10. Overall, TSB is the high-performing and secure prefetcher outperforming the timely secure version of all prefetchers by 4.1%.

Speedup with SUF. Next, we show the effect of SUF with and without timely secure prefetching (Fig. 11). The first (white) 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 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% performance degradation of GhostMinion (red line).

The next bar shows the speedup achieved by SUF, which improves the performance of all secure prefetchers with the



(a) SPEC CPU2017



(b) GAP

Fig. 12. Speedup of Berti, TSB, and TSB+SUF normalized to a non-secure system without prefetcher. The higher the better.

highest improvement of 3.7% for Bingo and the lowest of 1.9% for Berti. SUF improves the effectiveness of all the prefetchers irrespective of their timely secure versions. Overall, TSB with SUF outperforms all other prefetchers. TSB without SUF achieves the same speedup as secure Berti + SUF (23.0%). When SUF is added to TSB, the speedup increases by 4.2%, reaching a total of 28.4% (over baseline). Importantly, TSB+SUF mitigates the performance degradation of using a secure system from 5.1% (in a system without prefetching) to 2.1% (in a system with prefetching). Finally, TSB can also be applied to non-secure cache systems, thus removing any speculative side-channel attack induced by the prefetcher. In that case, TSB performs on par with respect to on-access Berti (speedup for TSB 1.310 (not shown in Fig. 11) vs. speedup for Berti 1.311).

Individual speedup. Fig. 12 shows the individual speedup for on-commit Berti, TSB, and TSB+SUF. For SPEC traces, TSB improves performance by more than 5% in 7 out of 45 traces (15.6% of all traces) when compared with the on-commit Berti. For *603.bwaves_s-2931B*, performance improves 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, which drops by 42.2%. TSB and TSB+SUF only sees a performance

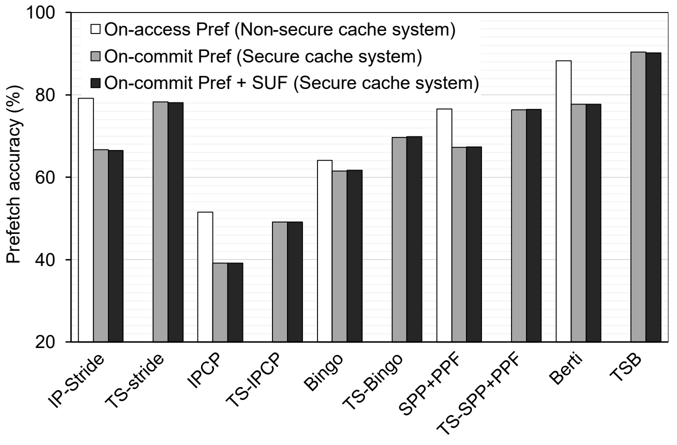


Fig. 13. Average prefetch accuracy. The higher the better.

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 of the SUF filter. Interestingly, TSB and TSB+SUF do not degrade performance in any trace. This is because, on average, SUF filters accurately for 99.3% of the time, with the maximum accuracy of 99.9% for `654.roms_s-1613B` and a minimum of 87.26% for `605.mcf_s-1554B`, improving the effectiveness of GhostMinion with prefetching.

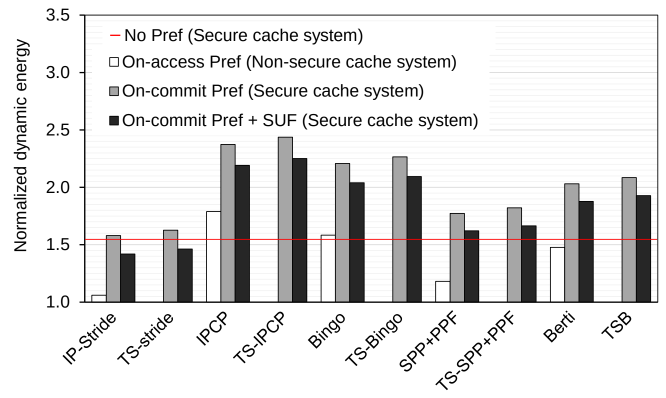


Fig. 14. Normalized dynamic energy consumption. The lower the better.

Prefetch accuracy. Fig. 13 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 accuracy with timely secure versions is visible. TS versions of IP-stride, IPCP, Bingo, SPP+PPF, and Berti improve prefetch accuracy, significantly.

Prefetch coverage. When transitioning from an on-access non-secure system to an on-commit secure system, the average MPKI increase across all prefetchers is 4.3%, 7.4%, and 8.3% at the L1D, L2, and LLC, respectively. As with accuracy, since SUF does not change the way that prefetchers work, its coverage remains the same for all of the prefetchers. All the timely secure prefetchers improve coverage by atleast $\approx$2%. TSB, achieves an MPKI reduction of 2.4%, 4.9%, and 8.8% at the L1D, L2, and LLC caches, respectively, compared to 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 correct latency seen by it, which provides the least late and incorrect prefetch requests.

Memory hierarchy traffic. GhostMinion adds traffic due to the writeback and re-fetch requests. Across all prefetchers, the average traffic increase compared to the on-access version is 54.7% for L1D, 46.6% for L2, and 40.4% for LLC. SUF mitigates the increase in traffic generated by GhostMinion with all the prefetchers.

L1, L2, and LLC access Latency. The utilization of a secure system cache increases the latency of all prefetchers at all levels, with a significant increase in the L1D cache. On average, across all the on-commit prefetchers, latency increases by 34.8% in the L1D cache, 5.2% in the L2 cache, and 8.8% in the LLC cache with respect to its on-access counterpart. Among 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. There is a direct correlation between traffic and dynamic energy consumption overhead in the memory hierarchy (Fig. 14). 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 as well as TS-stride are 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.

B. Multi-core performance

Fig. 15 shows the performance of SUF and TSB on a 4-core simulated system. The mixes have been sorted in increasing order of speedup. Compared to the non-secure baseline, GhostMinion incurs an average performance overhead of 16.8%. Only 14 mixes show a performance improvement. Similar to the single-core scenario, SUF does not improve performance significantly without a prefetcher when used in a multi-core system. In the case of a multi-core system with SUF turned ON, we observe a slight performance improvement of 0.9% over GM without SUF.

With prefetcher ON, 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. The SUF filter improves performance over a secure cache system in all mixes. TSB+SUF improves the performance over the non-secure baseline by 16.1%, followed by on-commit Berti+SUF (12.4%). When compared with secure on-commit Berti, TSB+SUF improves performance by 23%. Note that SUF's average accuracy drops marginally from 99.95% in single-core to 99.25% in a multi-core system. This marginal accuracy drop is because of the cross-core evictions at the shared LLC, and this drop does not affect overall performance.

SMT-based multi-core systems. The effectiveness of SUF is driven by its accuracy and in an SMT core, one thread can evict cache lines of other threads from both L1D and L2. When we apply SUF and TSB on a 2-way SMT-based multi-core processor, we find that the average accuracy is still over 99%. The reason is that, on average, it takes 200 cycles from the time a speculative load request is generated till it gets committed. This latency is as low as 46.93 cycles for `603.bwaves-2931B`. So, the probability of an eviction at L1/L2/LLC that can lead to a misprediction by SUF is extremely low, which is also evident from high accuracy. There are mixes with multiple copies of `605.mcf-1554B`, `cc-14B`, `bc-0B`, and `bc-5B` where

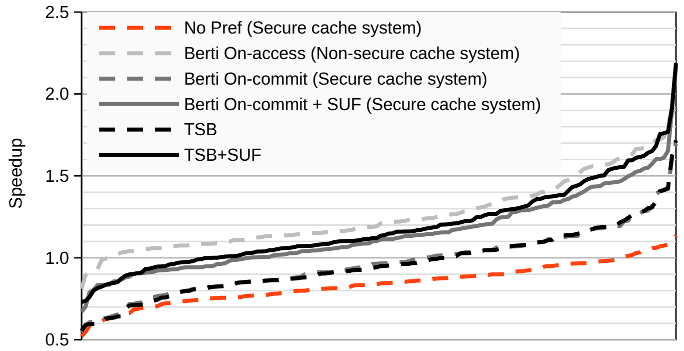


Fig. 15. Speedup for 4-core mixes normalized to the non-secure cache system with no prefetching.

the accuracy dropped to 91.74%. In summary, the effectiveness of SUF and SUF+TSB remains similar in multicore systems as shown in Figure 15 and also in SMT-based multicore systems.

VIII. Conclusion

Secure cache systems mitigate transient execution attacks like Spectre by providing strictness ordering. We showed that hardware prefetching is fundamental for hiding the performance overhead of a secure cache system. We performed, for the first time a comprehensive evaluation of the interaction between state-of-the-art prefetch mechanisms on a secure cache system. Our analysis shows that (i) state-of-the-art secure memory hierarchies prevent prefetchers from achieving their true potential and even in some cases turning significant speedups into no performance at all, and (ii) prefetching techniques perform sub-optimally when moving to commit due to loss of timeliness. We addressed these two problems and improved the effectiveness of hardware prefetchers. Our enhancements improve the effectiveness of a variety of prefetchers, with TSB being the best among all the secure prefetchers. In summary, our proposal improves the single-thread performance by 6.3% and the multi-core by 23.0% (over the top-performing Berti prefetcher) with 0.59 KB of storage overhead per core.

IX. Acknowledgements

We would like to thank members of the CASPER research group, especially Nishkarsh and Shubham. Thanks to Govind and Sawan for their feedback on the initial draft. This work is supported by the Qualcomm Faculty Award 2022 and Google India Research Award 2022. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 819134) and from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00).

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://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/, 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] 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.

[26] N. S. Kalani and B. Panda, "Instruction criticality based energy-efficient hardware data prefetching," *IEEE Comput. Archit. Lett.*, vol. 20, no. 2, pp. 146–149, 2021. [Online]. Available: https://doi.org/10.1109/LCA.2021.3117005

[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] A. B. Kvalsvik, P. Aimoniotis, S. Kaxiras, and M. Själander, "Doppelganger loads: A safe, complexity-effective optimization for secure speculation schemes," in *50thInt'l Symp. on Computer Architecture (ISCA)*, 2023, pp. 1–13.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 252–263.

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

[44] M. Sutherl, A. Kannan, and N. E. Jerger, "Not quite my tempo: Matching prefetches to memory access times," in *2nd Data Prefetching Championship*, Jun. 2015.

[45] 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.

[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.

[47] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, "Pensieve: Microarchitectural modeling for security evaluation," in *50thInt'l Symp. on Computer Architecture (ISCA)*, 2023, pp. 1–15.

[48] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *52ndInt'l Symp. on Microarchitecture (MICRO)*, 2019, pp. 954–968.