Mitigating transient execution attacks with focus on performance-security tradeoff

Bi-annual seminar-II 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)

20 June 2023

Table of Contents

List of Figures

Introduction

We address the threat model imposed by transient execution attacks, which exploit the speculative capabilities utilized in modern processors. Speculative execution is a powerful technique employed by processors to enhance performance by executing instructions ahead of time based on predictions. However, this technique can introduce security vulnerabilities when sensitive information is involved. To mitigate such attacks, hardware-based approaches like MuonTrap and GhostMinion have been proposed, minimizing the impact on performance.

While MuonTrap focuses on transient data sharing across domain boundaries, GhostMinion provides a more generalized solution to counter new variants of Spectre attacks. GhostMinion employs a filter cache, along with multiple restrictions, to defend against different attack variants by addressing the vulnerabilities associated with speculative execution.

We investigate and evaluate various prefetching techniques within the GhostMinion environment which ensures security. The performance impact of these prefetching variations is analyzed using IP-stride, IPCP, and Berti prefetchers.

The results and findings from these experiments provide insights into the effectiveness of different prefetching techniques in a secure environment with GhostMinion. The analysis considers factors such as performance degradation, cache pollution, and changes in the cache hierarchy. The goal is to identify the primary contributors to performance overhead and optimize the prefetching mechanisms accordingly. The observations from this study contribute to enhancing the security and performance of processors in the presence of transient execution attacks.

Threat Model

In our work, we are considering a threat model imposed due to transient execution attacks. Transient execution attacks exploit the speculative capabilities employed in most modern processors. Speculative execution is one of the most powerful techniques used in modern processors to boost performance by multiple folds. It is based on the principle of executing instructions ahead of time, based on predictions of the most likely execution path, to fully exploit all the idle micro-architecture resources at hand. However, this technique can introduce security risks when sensitive information is involved.

In this threat model the attacker tires to leak secret data by taking advantage of transient execution vulnerabilities. The attack can be similar to Spectre[1], where mis-speculation doesn't trigger explicit incorrect program behaviour, rather exceptional behavior is temporarily ignored during speculative execution. Then the attacker can bring in or evict data from observable caches beyond the point of misspeculation. Although the attacker does not necessarily need to probe the cache to get the secret data directly, rather recent attacks like SpectreRewind[2] have shown the ability to measure secrets by analyzing the effects on concurrently executing instructions which leads to long-lasting effects on the cache.

We can assume that the attacker does not have direct access to the secret data, as runtime checks are in-place to restrict the direct access. The attacker operates within a restircted environment where it can run some arbitrary code, sharing an address space with the secret. However, the inability to directly access secret data necessitates exploiting transient execution vulnerabilities to indiretly infer the sensetive information. The attacker could also reside in another process or another system entirely.

Mitigation strategies

Numerous mitigation strategies have been proposed to prevent Spectre attacks, which exploit speculative execution in modern processors. One approach is to stop speculative execution whenever code accesses secret data. However, current processors do not have the functionality to completely disable speculative execution. An alternative solution is to use instructions like "lfence" to block speculative instructions. However, this software-based method incurs high performance overhead and requires modifying all binaries.

Another mitigation technique is to prevent access to secret data by replacing bounds checking code with techniques like index masking, which eliminates conditional branch instructions. However, this approach also requires modifying software and suffers from the same drawback of needing changes to all binaries.

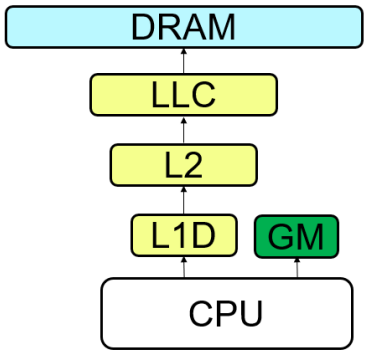


Figure 3.1: Modified cache hierarchy for GhostMinion

Recently, hardware-based approaches such as MuonTrap[3] and GhostMinion[4] have been proposed to mitigate Spectre attacks with minimal performance impact. MuonTrap addresses the issue of transient data sharing across domain boundaries, such as context switches between processes. It introduces a small cache called the filter cache, which stores speculative data and can be flushed in a single cycle upon a domain boundary switch. The rest of the cache hierarchy remains unaffected by speculative data requests, and only committed instructions write data back to the cache hierarchy.

New variants of Spectre attacks, such as SpectreRewind and Speculative Interference, exploit the timing effects of speculative instructions on logically earlier instructions. To mitigate these attacks, GhostMinion was proposed as a more generalized solution. Like MuonTrap, it uses a filter cache to filter speculative requests. However, GhostMinion employs multiple restrictions on the filter cache to defend against new attack variants.

GhostMinion's distinguishing factors include parallel access to the L1 cache, temporal ordering of instructions using timestamping, eviction of older instructions' data before newer instructions can access it, eviction of cache lines from the filter cache upon instruction commitment, and the ability for older instructions to evict newer instructions' MSHR entries to prevent backwards-in-time attacks. These techniques aim to mitigate a broader range of Spectre attacks by addressing the vulnerabilities associated with speculative execution in modern processors.

Prefetcher restrictions

Speculative memory accesses in the filter cache have the potential to change the cache state through prefetches. If prefetches are trained on speculative access stream, a prefetch triggered could potentially leak information to the broader cache hierarchy. Hence, just hiding the speculative memory access stream in the filter cache, which training prefetcher based on them, is insufficient to guarantee security.
GhostMinion suggests the following methods to operate prefetchers in a secure manner.

- **On-commit:** Prefetcher is unaware of the speculative access stream. Speculative memory accesses are tagged with the cache level they were brought in from. This information can be used on commit to train only those prefetches present at the corresponding level. As the prefetcher is trained based only on committed instruction, it will not be able to leak any information that could have been leaked otherwise.

- **Speculative prefetching to filter cache:** Prefetcher is trained on the speculative memory access stream, with the restriction that the prefetch blocks are filled directly into the filter cache and none of the prefetches goes into the non-speculative cache hierarchy. This again guarantess security as the prefetches triggered based on the speculative stream are not propagated to the non-speculative cache hierarchy and hence no information leakage.

Apart from the above prefetching methods, we also tested out the insecure variation where the prefetcher functions unmodified and is trained on the speculative access stream. Hence prefetches triggered are based on the speculative access stream and can be lead to a potential vulnerability. Although it does not guarantee security, we examined this insecure variation of prefetching to compare the performance degradation caused by the secure variations.
We also test the on-commit variation of prefetchers in a non-secure environment where there is no ghostMinion. We do this to observe the performance degradation caused by the on-commit prefetching mechanism. This will show whether the on-commit mechnism is

the major contributor to degradation in performance or the change of interactions in cache hierarchy imposed by GhostMinion.

Table 4.1 describes all the prefetching variations possible along with the environments they are active in. The naming convention follows the prefetching technique, followed by the environment (secure/non-secure). For example, L0D speculative(GM) is the prefetching technique where prefetches are trained and triggered on speculative access and work in the GhostMinion environment.

In the next section, we will discuss the performance impact of all the configurations mentioned in Table 4.1.

Configuration	**Description**
L0D speculative (GM)	Prefetcher is trained and triggered with speculative requests coming at the L0D cache. Prefetches are filled directly into L0D cache bypassing the non-speculative caches.
On-commit (GM)	Prefetcher is trained and triggered on committed instruction stream only. Prefetches are filled into non-speculative hierarchy without any restriction
L1D insecure (GM)	Prefetcher is trained and triggered with speculative requests coming at the L1D cache. Prefetches are filled into non-speculative hierarchy without any restriction. Populating speculative data in non-speculative caches makes it insecure
Normal (Non secure)	No modifications
On-commit (Non secure)	Prefetcher trained and triggered with committed instruction stream
Secure (GM)	Agustin implementation. Prefetcher trained and triggered on committed instruction stream. However, prefetcher stores useful information from speculative requests to train properly at commit time.

Table 4.1: Prefetcher configurations

Analyzing Prefetching Variations in GhostMinion

Figure 5.1 shows the performance impact of all the different prefetching variations described in table 4.1 when used with IP-stride, IPCP[8] and Berti[9] prefetcher. We have used a common baseline for all the experiments. The baseline used here has no prefetchers with a non-secure environment, i.e., no GhostMinion is used. This is done to have a uniform baseline for all the experiments.

As we can see from the plots, for a given prefetcher there is a lot of variability in performance across the various configurations. However, the performance trends across different configurations remain the same across different prefetchers. The Vanilla(no GM)

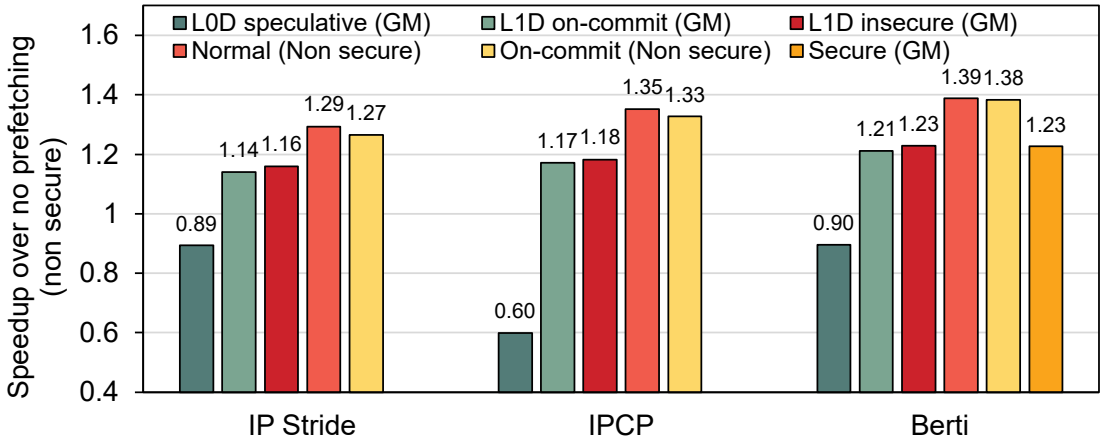


Figure 5.1: Speedup for SPECCPU 2017

bars shows the performance of prefetchers with their stock configuration in a non-secure environment with no GM. As seen in figures 5.1, Berti is the best performing prefetcher with a performance improvement of 39%, followed by IPCP and IP-stride. We want performance in a secure environment to be as close as possible to this improvement we get in the non-secure environment. All prefetchers perform poorly for the first configura-

tion, where the prefetcher is trained on the speculative access stream and the requests are brought into the filter cache directly. As we can see from Figure 5.1, all three prefetchers show no performance benefits and, in fact, suffer from significant performance degradation. The primary reason for this degradation is the small size of GhostMinion(GM). Due to the small size of GM cache, prefetches brought into it causes a lot of cache pollution by evicting demand blocks with high re-usability. Another major contributing factor to the degradation is the extra traffic brought in by the prefetch requests which puts pressure on micro-architecture structures like Miss-status holding register(MSHR) and various buffers in between the caching interfaces. This increases the response time of the demand requests, slowing down the entire system. This is more evident from the fact that IPCP being the most aggressive among the three prefetchers, incurs the highest performance hit. By an aggressive prefetcher we mean, a prefetcher that issues more prefetches with higher prefetch distance. We can conclude that this is not the best prefetching technique among the different techniques we have in hand.

The second bar of each prefetcher shows the performance improvement for its on-commit version. Berti has highest improvement of 21%. Alhtough it is improving performance, we can still see a gap of 18% compared to its vanilla version in non-secure environment. There might be two possible reasons for this gap in performance. First, it is possible that the on-commit mechanism is hindering the training of the prefetcher. Specifically, the speculative access pattern may train the prefetcher differently than the committed instruction stream. This is due to the access pattern being convoluted by the various delays incurred at the CPU and throughout the memory hierarchy till the commit point is reached. Second, it might be the case that major performance degrader is the changes in cache hierarchy brought in by GhostMinion. As discussed earlier, GhostMinion changes the way how data packets interact with the cache hierarchy in a significant way. Unlike a normal cache hierarchy where a block present in the DRAM, advances to the LLC, then to L2 and L1 and finally given back to the CPU, in a GhostMinion setting a speculative block from DRAM is directly transfered to the filter cache through the intermediate MSHRs of all the caches, without changing the state of the non-speculative cache hierarchy. And on commit of an instruction is the block transferred from the filter cache to L1. And then it finds it way back to L2 and LLC when evicted from L1. We have discused multiple experiments going forward to know which one is the primary reason for performance degradation.

We also tested the on-commit version used in the GhostMinion environment in a non-secure environment. This enabled us to determine if the on-commit mechanism itself was the primary cause of improper training of the prefetcher. As we can see from Figure

5.1, the on-commit version performs equally well as the vanilla versions of respective prefetchers in a non-secure environment. In the case of Berti, there is only a 1% performance degradation compared to the unmodified version of Berti. This indicates that the on-commit mechanism is not the major contributor to the performance degradation in the case of GM.

To further strengthen the fact, we examined how the vanilla version of each prefetcher work in the GM environment without any prefetching restriction. As seen from figure 5.1 , the vanilla version in GM performs almost similar to the on-commit version. Thus it is clear that moving from an unmodified version of a prefetcher to its on-commit version both in GM and non-secure environment, does not incur a significant performance penalty.

In the next section we will discuss further experiments to strengthen this fact and find out where exactly might the bottleneck might be present.

Performance analysis

This section discusses the potential causes of performance overhead. We will examine problems on a case-by-case basis, beginning with a simple IP-stride prefetcher and exploring potential issues in the training process. We will suggest simple tweaks that can be implemented to improve performance. Next, we will examine a more sophisticated hardware prefetcher, the Berti prefetcher, and perform the same analysis as with the IP-stride prefetcher.

IP-stride: Stride-based prefetching is a straightforward approach to prefetching, wherein the prefetcher aims to identify consistent patterns in cache accesses. One example of this technique is IP-stride prefetching, which associates a specific stride with a particular program counter (PC). The prefetcher learns the stride by calculating the difference between the current address and the previously recorded address linked to that specific PC. As the same PC exhibits a recurring stride, the confidence in the learned pattern increases. Once the confidence surpasses a predefined threshold, prefetches are initiated for the corresponding PC using the acquired stride.

The on-commit variation is the best performing prefetching variation of IP-stride in the GM environment. Therefore, we will proceed with it. When we move prefetcher training and issuing prefetches to on-commit, it hinders the training of IP-stride. This is because there is a large gap between the time a request comes to the cache and when it is committed. The access pattern seen by the IP-stride prefetcher does not resemble the actual access pattern, and training is hampered as a result.

To understand how the training is hindered we looked at the lateness numbers. Before going into the results, let us discuss what lateness is. A prefetch is said to be late, if it was present in the MSHR while a demand request asked for the same address. If the prefetch would have been in time, the demand request would have been convereted from a miss to a hit, hence, saving cpu cycles. We call this just “late” prefetches. Figure 6.1 shows the percentage of late prefetches for IP-stride in a non-secure environment. The mean lateness percentage is 7 %.

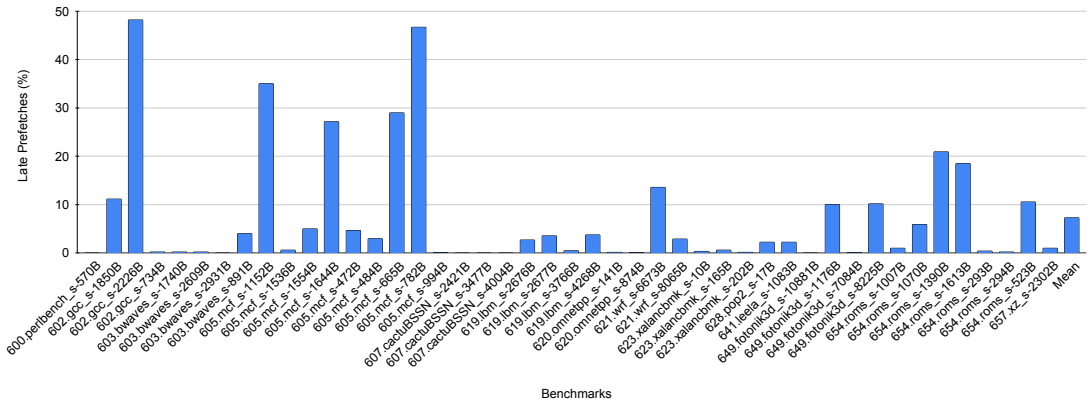


Figure 6.1: Lateness in IP-stride prefetcher

However, in a GhostMinion environment, just quantifying these late prefetches is not enough. The reason being, a prefetch request triggered on commit that could have been triggered on access, if prefetching restrictions were not there, could have converted more demand misses into demand hits. Hence these prefetches should also be categorized as late prefetches. We call these late prefetches as commit late. There can be another type of late prefetch, where a prefetch request is in the MSHR but it could have been triggered early if the prefetches were triggered on access.

Figure 6.2 shows the lateness number for on-commit version of IP-stride functioning in a GhostMinion environment. As we can see the percentage of late prefetches increases only by 4% .Also, the commit late prefetches only account for a fraction of the total late prefetches.

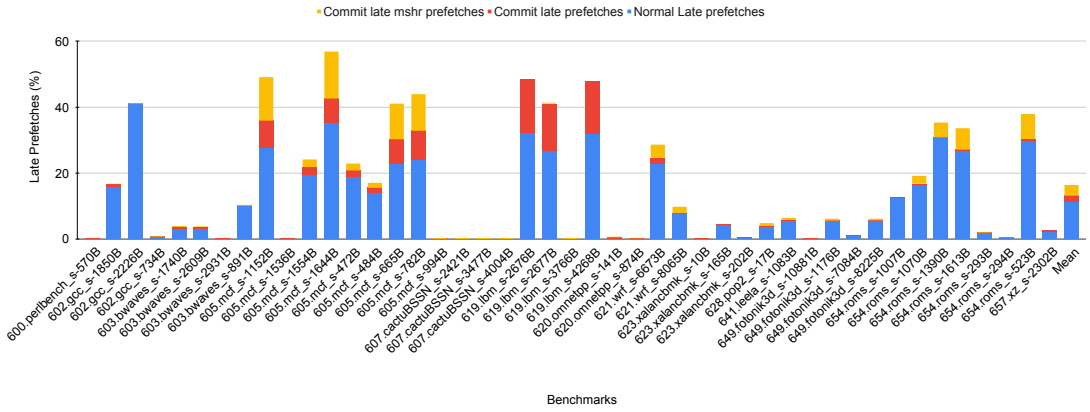


Figure 6.2: Commit late prefetches in IP-stride prefetcher

To counter the effect of lateness, we performed another set of experiments where we increased the prefetch distance of all prefetches issued by the IP-stride prefetcher by a

fixed multiple. Figure 6.3 shows the change in performance of on-commit IP-stride when we change the prefetch distance by multiple of 4, 8, 16 and 32.
Benchmarks like gcc, mcf, xalancbmk, wrf improves performance with higher prefetch distance as they have a higher resuse distance. But again, apart from a few benchmarks, there is no significant performance improvement accross all benchmarks. We can conclude that timeliness is not the issue causing performance degradation in case of on-commit prefetching in a GhostMinion environment.

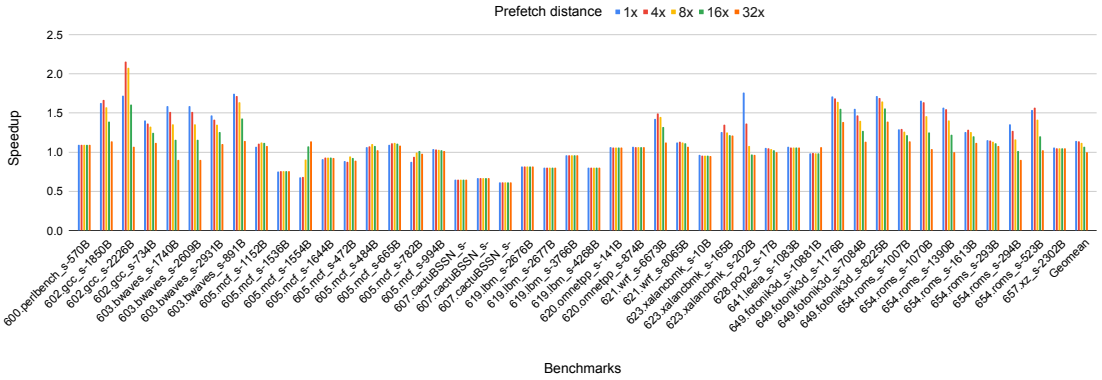


Figure 6.3: Effect of Distance on IP-stride prefetcher

Berti is an advanced prefetcher designed to achieve high accuracy by selecting the most effective deltas that can trigger timely prefetching, tailored to each instruction-pointer (IP). In this context, a delta refers to the difference between two cache line addresses. Unlike traditional approaches that determine the best delta across all accesses, Berti localizes this optimization process to each unique IP. This localized approach significantly enhances accuracy, leading to improved performance.

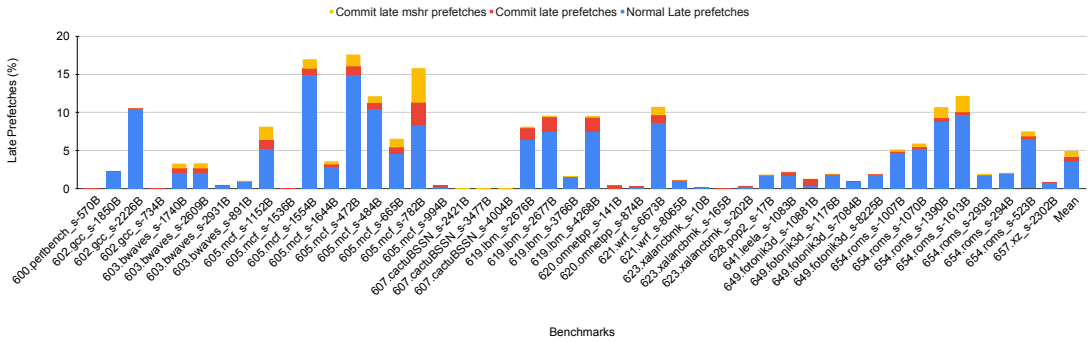
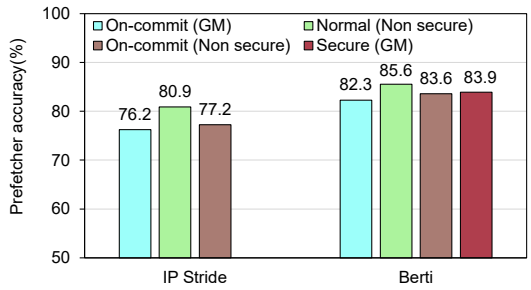
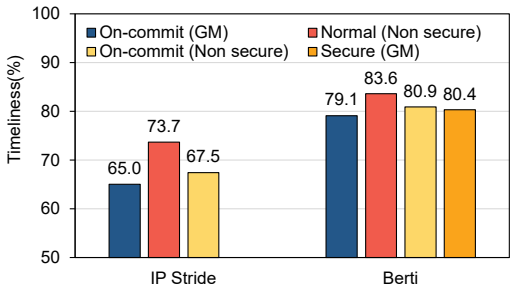


Figure 6.4: Commit late prefetches in Berti prefetcher

Figure 6.4 shows the lateness number for on-commit version of Berti functioning in a GhostMinion environment, similar to IP-stride. As we can see the percentage of late



(a) Prefetch accuracy including late prefetches



(b) Timely prefetches

prefetches is 5% overall. Also, the commit late prefetches only account for a fraction of the total late prefetches, which is similar to IP-stride. Compared to the unmodified version of Berti running in a non-secure environment, the overall lateness only increaes by 3%. This again enforces the fact that lateness is not the primary issue causing performance degradation.

Figure 6.5a, 6.5b shows the prefetcher accuracy and timeliness of all the prefetching variations of IP-stride and Berti over all the SPECCPU 2017 benchmarks.

As discussed in the mitigation section, GhostMinion changes the flow of data in the cache hierarchy when compared to the normal cache hierarchy. In GM, a speculative data is filled directly to the filter cache, without changing intermediate cache states. The speculative data is written back to the L1 cache when the instruction associated with the data is committed. Unlike the normal cache hierarchy where data flows from the DRAM towards the upper level caches, the flow of data is reversed. It flows from the filter cache to the rest of the non-speculative caches and then to the DRAM.

We believe that this is the major factor causing performance degradation in GM. We are still in the process of pinpointing the exact reasons by performing further analysis of different metrics.

Improving Berti

In this section we discuss modifications made to Berti. First we discuss the changes to make it work with the on-commit instruction stream. Then we discuss about two modifications made to the on-commit version to further improve performance. Berfore discussing about the modifications, let us briefly look at how Berti works, and the basic principle behind its functioning.

Berti, a local data prefetcher, operates with a focus on timely prefetching by using deltas instead of strides to understand access patterns. Deltas refer to the difference between two cache line addresses. Berti's goal is to learn the specific delta that can effectively trigger a prefetch operation for each instruction pointer (IP).

In the provided example shown in Figure 7.1, Berti aims to cover the address 12. To achieve this, Berti observes that the fill latency for address 12 is 50 cycles, meaning that only an access occurring before cycle 20 can initiate a prefetch operation in a timely manner. In this particular case, only the access to address 2 can serve as a trigger for a timely prefetch.

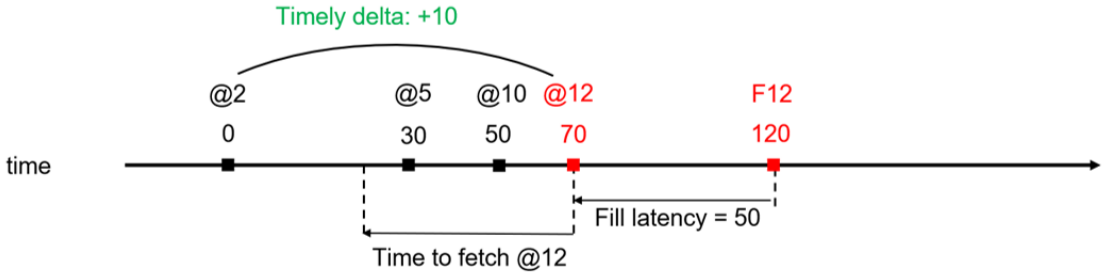


Figure 7.1: Notion of timely delta

Now, let's briefly discuss how Berti is trained and how it triggers prefetches.

A history table is utilized to store previous cache accesses in Berti's approach. Each entry in the table includes the IP (instruction pointer), cache line address, and the timestamp of the last access. Berti uses this information to calculate the fill latency for cache misses and prefetch fills. Prefetches are also taken into account during the training process, treating

them as misses if the prefetcher was not present.
Once the fill latency is determined, Berti examines the past access history in the history table to identify timely deltas that can trigger timely prefetches. These timely deltas are recorded in the delta table, along with their associated coverage. Coverage represents a confidence counter that indicates the number of times a specific delta has been selected for a particular IP out of the total number of timely deltas chosen for that IP.
Using the information in the delta table, the prefetcher issues prefetches by adding the most appropriate delta to the current cache line address. The prefetching is performed up to a certain cache level, which is determined based on the coverage value of the respective delta.
To make Berti work with the committed instruction stream, we extend the ROB to store information about speculative requests that can be used at commit time to train the prefetcher properly. ROB is extended with three extra fields: cycle, hit/miss flag, prefetch flag. In the presence of GhostMinion, an extended Reorder Buffer (ROB) is used to store information about L1D cache accesses. When there is an access in L1D, the extended ROB records the cycle at which the access occurs, whether it was a hit or a miss (indicated by a hit/miss flag), and if it was a hit due to prefetching (prefetch flag). During the commit phase of a LOAD instruction, Berti undergoes training using the data from the extended ROB, just as in the normal process. However, there is a slight modification in calculating latency. The latency is determined by subtracting the saved cycle from the current cycle. This latency includes the fill latency (time taken to fill the cache) and the retirement latency of the ROB. In the same step, Berti also issues a prefetch based on the acquired information.
A potential issue arises with writes in this implementation. It is possible for writes to be retired from the ROB before the corresponding cache block has been filled in the L1D. To ensure proper training of Berti, the following approach is taken: if the cache block associated with a write is still in-flight (not yet filled) when the write is retired, the address of that write is saved in a buffer. Once the cache block corresponding to the saved address is eventually filled, the latency is calculated, and Berti is trained accordingly.
On top of this, we implement two changes to further improve performance.
When filling L1D cache on commit, there is a change in the temporal behavior of the replacement algorithm, leading to an increase in early prefetchers. This can be observed in scenarios like the "mcf" benchmark, where using GhostMinion (GM) results in issuing twice the number of prefetch requests compared to the case without prefetching. Consequently, this increase in prefetch requests leads to higher latency and an elevated CPI (cycles per instruction), despite having a lower MPKI (misses per kilo-instructions).

To address this issue, a potential solution is to train and issue prefetches using L0D (a lower-level cache) but store the prefetch requests in a buffer. When cache lines are eventually moved from L0D to L1D, any prefetch requests that match demand requests can also be transferred. The intention behind this approach is to maintain a replacement algorithm's temporal access pattern as closely as possible to the scenario without GhostMinion, thereby mitigating the negative impact on performance caused by the increased number of prefetch requests.

The next idea involves the use of a shadow cache to prevent issuing prefetch requests to L2 cache for blocks that are already present in L1 cache. In the on-commit Berti approach, the only notable difference is a small modification: we no longer train Berti twice when encountering consecutive misses on the same cache line. This is done, as in a GhostMinion environment, the blocks present in L1 cache will be moved to L2 and LLC eventually on eviction.

Both ideas improve the performance of on-commit Berti marginally. However, when we change the baseline to a configuration with no prefetcher and GhostMinion, we observe a considerable performance improvement of 33%. This is because GhostMinion itself significantly degrades performance. And ,we can infer that prefetchers are already performing at their best. Therefore, the problem is how we can alter the interactions in the cache hierarchy to minimize performance degradation.

Conclusion and Future work

We examined the performance impact of different prefetching variations in the presence of GhostMinion, a generalized solution designed to mitigate Spectre attacks. Our experiments revealed notable performance variations across different prefetching techniques and configurations.

Our analysis demonstrated that the on-commit prefetching variations, both in the GhostMinion environment and in a non-secure environment, did not significantly impact performance compared to their unmodified counterparts. This indicated that the on-commit mechanism itself was not the primary cause of performance degradation in the GhostMinion setting.

Further investigation revealed that the major contributor to performance degradation in the GhostMinion environment was the changes in the cache hierarchy induced by GhostMinion. The reversed data flow, where speculative data is filled directly into the filter cache and written back to the L1 cache upon instruction commitment, disrupted the normal cache access pattern.

The changes introduced by GhostMinion in the cache hierarchy, along with other factors, significantly influenced performance. Further analysis and optimizations are required to address these challenges and improve the effectiveness of hardware-based mitigations for transient execution attacks.

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
20 June 2023

References

[1] Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg et al. "Spectre attacks: Exploiting speculative execution." Communications of the ACM 63, no. 7 (2020): 93-101

[2] Fustos, Jacob, Michael Bechtel, and Heechul Yun. "SpectreRewind: Leaking secrets to past instructions." In Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security, pp. 117-126. 2020.

[3] Ainsworth, Sam, and Timothy M. Jones. "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state." In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 132-144. IEEE, 2020

[4] Ainsworth, Sam. "GhostMinion: A strictness-ordered cache system for Spectre mitigation." In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 592-606. 2021

[5] Y. Yarom and K. Falkner, “Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in USENIX Security Symposium, 2014

[6] Liu, Fangfei, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-level cache side-channel attacks are practical." In 2015 IEEE symposium on security and privacy, pp. 605-622. IEEE, 2015

[7] Behnia, Mohammad, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer et al. "Speculative interference attacks: Breaking invisible speculation schemes." In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1046-1060. 2021

[8] Pakalapati, Samuel, and Biswabandan Panda. "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching." In 2020 ACM/IEEE

47th Annual International Symposium on Computer Architecture (ISCA), pp. 118-131. IEEE, 2020.

[9] Navarro-Torres, Agustín, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibánez, Víctor Viñals-Yúfera, and Alberto Ros. "Berti: an Accurate Local-Delta Data Prefetcher." In 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 975-991. IEEE, 2022.