SpecPref: High Performing Speculative Attacks Resilient Hardware Prefetchers

Tarun Solanki
Dept. of CSE
Indian Institute of Technology Kanpur
taruns@cse.iitk.ac.in

Biswabandan Panda
Dept. of CSE
Indian Institute of Technology Bombay
biswa@cse.iitb.ac.in

***Abstract*—With the inception of the Spectre attack in 2018, microarchitecture mitigation strategies propose secure cache hierarchies that do not leak the speculative state. A recent mitigation strategy, MuonTrap, proposes an efficient, secure cache hierarchy that provides speculative attack resiliency with minimum performance slowdown. Hardware prefetchers play a significant role in improving application performance by fetching and bringing data and instructions into caches before time. To prevent hardware prefetchers from leaking information about the speculative blocks brought into the cache, MuonTrap trains and triggers hardware prefetchers on the committed instruction streams, eliminating speculative state leakage. We find that on-commit prefetching can lead to significant performance slowdown as high as 20.46% (primarily because of prefetch timeliness issues), making hardware prefetchers less effective. We propose Speculative yet Secure Prefetching (SpecPref), enhancements on top of the MuonTrap hierarchy that allows prefetching both on-commit and speculatively. We focus on improving the performance slowdown with the state-of-the-art hardware prefetchers without compromising the security guarantee provided by the MuonTrap implementation and provide an average performance slowdown of 1.17%.**

I. Introduction

With the advent of the Spectre attack [13] in 2018, and other speculative-execution attacks [19]–[22], there is a pressing need to build mitigation strategies against these attacks that not only provides security but also remains concerned with the performance overhead. Amongst the various proposed mitigation techniques [8], [23]–[27], a recent proposal, named MuonTrap [3], provides a cache hierarchy that is resilient to speculative attacks and at the same time does not incur significant performance overhead. It also discusses hardware prefetching, which is not the case with other prior works. MuonTrap prevents speculative information leakage by adding a small speculative L0 cache between the core and the L1 cache. MuonTrap forces all the speculative data and instructions to reside only in the L0 cache, meaning that other caches in the cache hierarchy contain non-speculative data and instructions. When a memory instruction commits, the corresponding data and the instruction blocks are written to respective L1D and L1I caches. MuonTrap uses a *committed* bit per cache block at the L0 to differentiate speculative blocks from the non-speculative ones. For the rest of the paper, we refer to the L0 cache as *speculative filter cache* and the rest in the hierarchy as *non-speculative caches*. The speculative filter caches (L0D and L0I) get flushed out on context and protection domain switches. This does not let any speculative data leakage between the processes.

A recent speculative interference attack [29] demonstrates MuonTrap to be vulnerable through miss status holding registers (MSHRs) and execution unit (EU) port contention. To mitigate this attack, the paper also discusses a few basic and advanced defense mechanisms. The advanced defense approach proposes tagging the instructions with a priority based on their age in the reorder buffer (ROB). This does not allow the younger speculative instructions to influence the older non-speculative instructions, thus eradicating the interference attacks. We thus improve the MuonTrap baseline and make it stronger in security by implementing the proposed advanced defense technique.

Hardware prefetchers play a significant role in improving application performance by fetching and bringing data and instructions into caches before time. To prevent hardware prefetchers from leaking information about the speculative blocks brought into the cache, MuonTrap allows the invocation of hardware prefetcher only after the corresponding instruction commits. Not only does it delay the prefetch training, but it also causes the upcoming prefetch blocks to become useless because of their late arrival into the cache, thus degrading the performance.

Ghost Loads [28] solves this problem by prefetching *speculatively* into a small speculative buffer (ghost buffer). However, Ghost loads only considers conventional data prefetchers like stride-based. Additionally, it does not consider the prefetch degree (number of prefetch requests issued at a given time) and distance (how far ahead of the demand access, prefetch requests are issued); which are extremely important with the state-of-the-art data prefetchers [9], [11], [12]. Similar to MuonTrap, it uses a speculative buffer, which is extremely small (eight-entry) at L1D, which is ineffective for the aggressive prefetchers with the recent deep and wide processor cores (e.g., Intel's Sunny Cove) with 352 entry ROB [4]. Furthermore, the Ghost loads approach is specific to few memory consistency mechanisms like the release consistency (RC), whereas MuonTrap is generic in terms of memory consistency mechanisms.

Figure 1 shows the average and maximum performance slowdowns with *on-commit and secure* prefetching (as mentioned in MuonTrap), Ghost Loads and our proposal (SpecPref) with the state-of-the-art instruction and data prefetching techniques, compared to *insecure* prefetching techniques that prefetch even during the speculative execution. We use the approach outlined

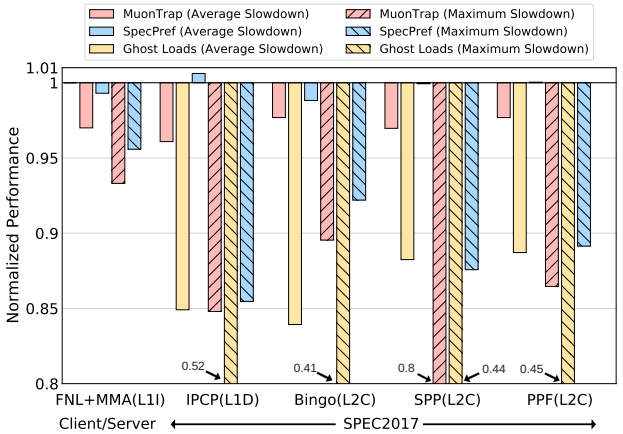


Fig. 1: Average and maximum performance slowdown with MuonTrap, Ghost Loads, and SpecPref (our proposal) with five state-of-the-art prefetchers compared to an insecure baseline evaluated with 50 client/sever [1] and 46 SPEC 2017 benchmark traces [2]. Bar above 1.0 represent performance improvement. Benchmarks with maximum slowdowns are different for SpecPref, Ghost loads, and MuonTrap. Ghost loads is not proposed for instruction prefetching.

in Ghost Loads [28] to identify whether an instruction is speculative or not. Ghost Loads uses a buffer and inserts an instruction that causes speculation (control, Stores, Loads, Exceptions causing instructions). If any load is younger than the oldest entry in the buffer, it is considered as a speculative load.

We discuss the state-of-the-art prefetching techniques in Section II. Please refer Table II for the simulated parameters. To understand each prefetcher in detail, we show performance slowdowns of all the prefetchers in isolation. At the L1-Instruction (L1I) cache, we consider the Footprint Next Line and Multiple Miss Ahead (FNL+MMA) [7] prefetcher, for server/client benchmarks, provided by the 1st instruction prefetching championship co-located with ISCA 2020 [17]. We consider Instruction Pointer Classification based spatial Prefetching (IPCP) [9] at the L1-Data (L1D) cache. At the L2 cache (L2C), we consider Bingo [10], Signature Path Prefetching (SPP) [11], and Perceptron based Prefetch Filtering (PPF) [12]. We evaluate data prefetchers at L1D and L2C using SPEC2017 [16] memory intensive benchmarks, provided by the 3rd data prefetching championship co-located with ISCA 2019 [5].

On average (maximum), FNL+MMA, IPCP, Bingo, SPP, and PPF show performance slowdowns of 2.96% (6.69%), 3.91% (15.20%), 2.31% (10.46%), 3.03% (20.46%), and 2.31% (13.54%), respectively with the MuonTrap implementation. Correspondingly, the maximum slowdowns are for `server_026`, `mcf`, `xalancbmk`, `mcf`, and `xalancbmk` benchmarks. Considering 2KB speculative buffers with Ghost Loads, the slowdowns on average (maximum) for IPCP, Bingo, SPP, and PPF are 18.23% (47.80%), 16.07% (58.65%), 11.76% (55.47%), and 11.29% (54.47%) respectively. For IPCP, the maximum slowdowns are with `bwaves`, and for the rest, it is the `fotonik` benchmark.

TABLE I: Increase in prefetch lateness and decrease in prefetch coverage due to on-commit prefetching.

	Lateness		Coverage	
	Average	Maximum	Average	Maximum
FNL+MMA	4.31%	8.04%	17.02%	30.98%
IPCP	2.39%	18.53%	3.62%	18.11%
Bingo	1.16%	7.69%	2.05%	10.15%
SPP	2.64%	22.61%	2.69%	16.06%
PPF	2.08%	17.35%	2.38%	11.89%

What causes this performance slowdown? One of the primary reasons for high performance slowdowns is the prefetch timeliness. Table I shows the average and maximum increase in prefetch lateness (prefetched blocks do not reach caches on time) and decrease in corresponding prefetch coverage (reduction in cache misses because of prefetching). One of the trivial approaches to solve the prefetch timeliness problem is to increase the prefetch degree and distance so that the prefetch requests will be triggered well ahead of time, with an aggressive prefetcher. We sweep through all possible combinations of prefetch degree and distance for all the evaluated prefetchers. With aggressive prefetching, for FNL+MMA and IPCP, we get a performance improvement of only 0.70% and 0.40%, respectively. Bingo prefetches cache blocks based on a memory region; thus, it has little scope for increasing the prefetch degree and distance. For SPP and PPF, we already use an aggressive version, and making it more aggressive by decreasing various threshold values does not further improve the performance.

Our contributions: In this paper, we propose Speculative yet Secure Prefetching (SpecPref), an enhancement to state-of-the-art hardware instruction and data prefetchers that allows speculative prefetching into speculative caches and on-commit prefetching into the non-speculative caches (Section III).

SpecPref incurs marginal changes to the cache hierarchy. When compared to insecure prefetching (that prefetches during speculative execution), it provides average performance slowdown of 1.17% with the different state-of-the-art hardware instruction and data prefetchers (Section IV).

II. Background

This section provides background on one instruction prefetcher (FNL+MMA) and four data prefetchers.

FNL+MMA: FNL+MMA [7] performs intelligent next-line instruction prefetching in its FNL component by predicting whether a cache line will be used in a reasonable future or not by monitoring the L1I accesses. This not only increases the performance but also addresses the wasted bandwidth and the cache pollution induced by the real next-line prefetching.

Applications also tend to have control-flow instructions, which renders jumps to the other locations to access the instruction blocks. These blocks, unable to be predicted by the next-line component, are taken care of by the MMA component of the FNL+MMA prefetcher. The MMA component associates two different cache blocks separated by an ahead distance to trigger timely prefetch requests. An ahead distance of n refers to the n^{th} block that will miss in the L1I-shadow cache. FNL+MMA placed 2nd in the instruction prefetching

championship [17]. We use FNL+MMA over the winner of the instruction prefetching championship, EIP [6], as FNL+MMA is equally effective with a significant low storage budget.

IPCP: IPCP [9] correlates instruction pointers (IPs) with the spatial access patterns. The spatial data prefetchers exploit the fact that the data block accesses are spatially correlated over small-sized memory regions, and this correlation can be used for prediction. IPCP follows a mechanism of classifying the IPs into three classes and designs tiny prefetchers per class. All the IPs appearing in the instruction stream can be classified into these three classes, and the resulting classification can be used for finer prefetching. At a given point of time, an IP can belong to one or many classes; hence a hierarchical priority is used for tie-breaking.

Bingo: The principal idea of Bingo [10] is inspired by the state-of-the-art branch predictor, TAGE [14]. The Bingo prefetcher implements spatial data prefetching by considering the short as well as the long events. The short events consider only some specific events for triggering a prefetch, for instance, the instruction pointer (IP). On the other hand, the long events consider the occurrence of various specific events like the combination of both IP and Address. This allows the long events to maintain higher prefetch accuracy. Though short events provide low prefetch accuracy, however, it has a higher chance of recurrence, which lets the Bingo prefetcher not lose any prefetching opportunity.

The footprint of the pages accessed by the CPU is associated with both the long and the short events and is consequently stored in the prefetcher metadata tables. The prefetch triggering occurs by looking up the footprint associated with the longest available event in the table, subsequently issuing the prefetch request. Bingo merges the multiple metadata tables into a single unified table; this significantly helps in reducing the storage overhead.

SPP: SPP [11] is a confidence-based lookahead prefetcher. The lookahead prefetchers prefetch blocks by speculating the upcoming blocks based on a delta value. A delta value corresponds to the difference between the block addresses; for instance, a delta of +3 signifies that block B+3 should be prefetched after block B has been accessed. To ensure the accuracy of the prefetches, SPP relies on the confidence mechanism based on its prior prefetch requests.

SPP creates a signature based on the history of accesses in a page and stores them in a metadata table. It then utilizes these signatures to predict the future likely delta patterns and thus issues the prefetch requests based on this learning. SPP also applies the history information on the new pages that occur during the demand accesses from the CPU; this allows SPP to prefetch even when a new page is accessed. SPP also changes the prefetching depth dynamically to maintain the prefetch coverage and accuracy.

PPF: Based on the underlying prefetcher SPP, PPF [12] aims to increase the prefetch accuracy by still maintaining the prefetch coverage provided by SPP. It sits between the underlying prefetcher and the prefetch queue, filtering out the inaccurate prefetch requests generated by SPP. To implement the mechanism of filtering, it uses the perceptron based learning that has been used in prior techniques as well, for instance, branch prediction. This filter mechanism can also be adapted by other prefetchers with some modifications.

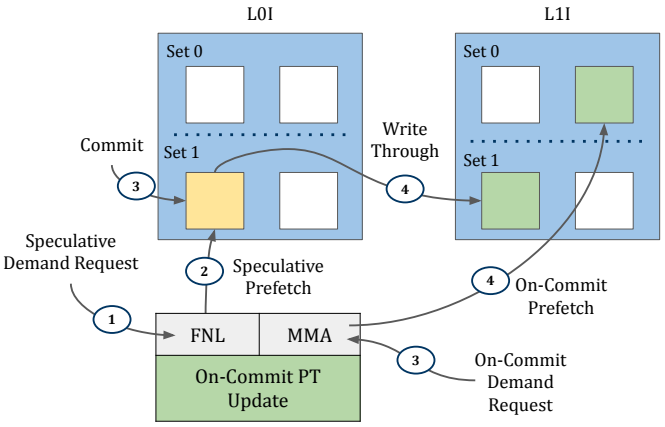


Fig. 2: SpecPref with L0I/L1I prefetcher.

III. SpecPref

In this section, we discuss our enhancements for instruction and data prefetchers, which we call SpecPref. We bring prefetch blocks during speculative execution into the speculative L0 cache, whereas for L1 and L2, we bring prefetched blocks on commit. For L2, we introduce a tiny speculative prefetch cache (similar to L0 for L1) for storing prefetched blocks brought during the speculative execution. All the prefetch tables (micro-architectural structures used for training) get updated only on *commit*. All the speculative structures like L0I, L0D, corresponding prefetch queues, miss status holding registers (MSHRs), and the newly introduced speculative prefetch cache get flushed on a context switch to mitigate the possibility of a transient execution attack. Note that, SpecPref follows the cache coherence mechanism similar to that of baseline MuonTrap implementation, and we do not propose any additional changes to coherence protocol on top of Muontrap.

A. L0/L1 instruction prefetcher

Figure 2 shows the steps in the implementation of SpecPref for the L1I prefetcher (which is now placed beside the L0I cache). In step ①, the speculative request at the L0I cache invokes only the FNL component. The MMA component can be used on commit with a large ahead distance. However, the FNL component fetches next lines, and if we do use FNL on commit, there will be no performance utility in terms of instruction prefetching. In step ②, the FNL component prefetches the speculative blocks into the L0I cache, bypassing the lower levels of the memory hierarchy (same as MuonTrap). Since the L0I cache is non-inclusive, this does not let any speculative state leakage into the non-speculative lower level caches, thus making the speculative state invisible to the entire cache hierarchy.

In step ③, when the commit of the corresponding instruction arrives, the committed blocks are written through (step ④)

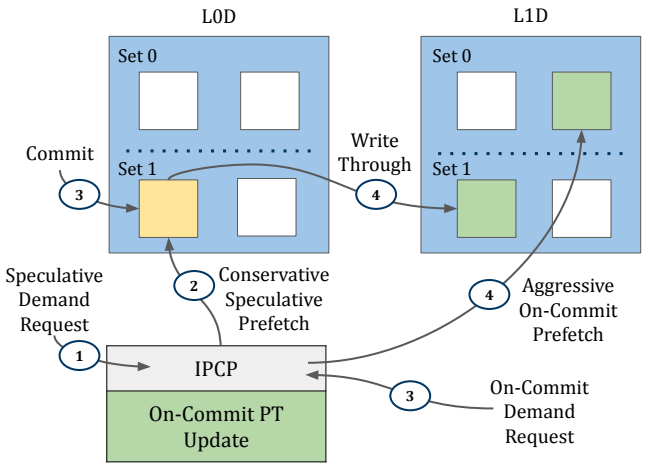


Fig. 3: SpecPref with L0D/L1D prefetcher.

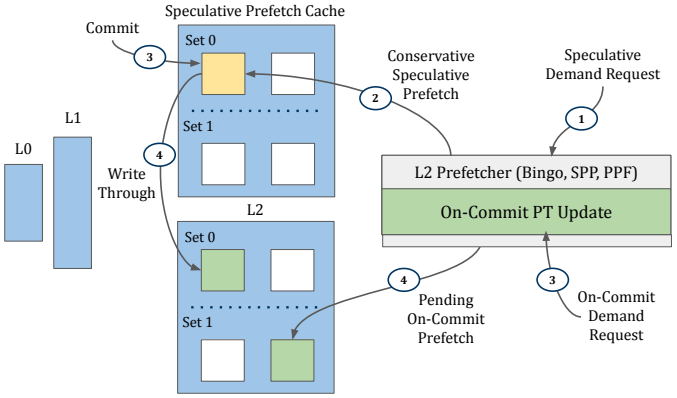


Fig. 4: SpecPref with L2 prefetcher.

from the L0I to the L1I, inclusively (making it visible to the entire cache hierarchy). Note that we do not write through the prefetched blocks as these blocks have *committed bit* as zero. However, if a committed instruction gets a hit on a prefetched block, then we set the committed bit and write through to the next level. The MMA component uses *ahead distance* that manipulates the distance value for the next predicted block w.r.t. to the prefetch triggering block. We reduce the prefetch lateness induced by the MMA component by increasing the *ahead distance*, and continue to prefetch the blocks only after the commit of the instruction. A high distance compensates for the lateness caused due to on-commit triggering of the prefetches of the MMA component. We use a lookahead distance of 16 for MMA, which used to be nine in the originally proposed FNL+MMA. we allow the updates to these tables only after the commit of the corresponding instruction.

B. L0/L1 data prefetcher

Figure 3 shows the steps in the implementation of SpecPref for the L1-D prefetcher (which is now placed beside L0-D). In step ①, the speculative request at the L0D invokes the IPCP prefetcher. In step ②, we allow only a conservative number of blocks (degree=1) to be prefetched into the L0D, bypassing the lower levels of the memory hierarchy. This reduces the possibility of L0D thrashing and prohibits any speculative state from leaking into the memory hierarchy. In step ③, when the corresponding load instruction commits, the committed blocks are written through (step ④) from the L0D to the L1D, inclusively (making it visible to the entire cache hierarchy). Note that we do not write through the prefetched blocks until they become committed, as discussed for the L0I prefetcher. Also, on the commit of the instruction, we issue prefetch requests (aggressively with a higher degree as suggested in the IPCP paper), which are filled into the L1D. As discussed for L1I, we allow only on-commit updation to the prefetcher tables.

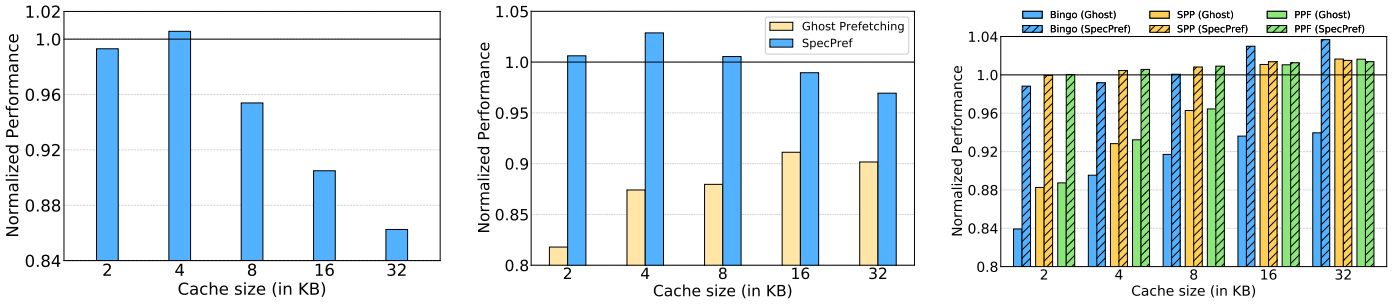
C. L2 data prefetchers

The design of L2 prefetchers such as SPP and PPF is such that they perform well with L1 misses, capturing irregular deltas that are filtered after L1 misses. To make L2 prefetchers effective, we invoke L2 prefetchers on an L1 miss. Figure 4 shows the steps in the implementation of SpecPref for L2 prefetcher; the steps are similar to those of the L1D prefetcher. We introduce a Speculative Prefetch Cache (SPC) that is located beside the L2 and probed concurrently. On a speculative L1 miss (step ①), the L2 prefetcher prefetches the speculative blocks into the SPC (step ②), bypassing the lower levels of the cache hierarchy. This does not let any speculative state leak into the non-speculative caches. At the commit point (step ③), the blocks from SPC are written through to the L2 (step ④), thus becoming visible in the memory hierarchy; subsequently, we trigger on-commit prefetching and bring pending blocks into the L2. Note that the prefetcher learning happens only on the commit access streams. At the SPC, too, we do not write through the prefetched blocks as discussed for L0I and L0D prefetchers, unless they get committed.

SPP and PPF also issue prefetch requests to LLC. We discard that mechanism (while prefetching speculatively) of these prefetchers and instead issue all prefetches only into the SPC. We use the degree and distance as per the look-ahead mechanism of SPP, which is dynamic in nature. For some benchmarks (`bwaves`), Bingo performs better with on-commit prefetching, thus we dynamically decide the mechanism between SpecPref and on-commit (using prefetch coverage as a metric) to prefetch the data. We quantify prefetch coverage for an epoch of L2 demand accesses (equivalent to the size of L2 in terms of blocks) with on-commit and SpecPref prefetching, and use the one that provides better coverage. Analogous to the L0, we flush, SPC on context and protection domain switches. For L2 prefetchers, we use a tiny 2KB (4 way, access latency of one cycle) of SPC. Note that Ghost Loads uses a 16KB speculative cache [28].

IV. Evaluation

We use the ChampSim [15] simulator with the MuonTrap cache hierarchy as mentioned in the Table II. We simulate 46 memory intensive benchmarks from SPEC2017 [16] for the L1D and L2 prefetchers. For the L1I prefetcher, we simulate 50 client/server traces provided by the 1st Instruction Prefetching Championship [17].



(a) L0I Cache (b) L0D Cache (c) Speculative Prefetch Cache at L2

Fig. 5: Size of speculative caches and average normalized performance with SpecPref. MuonTrap uses 2KB of L0D and L0I. Higher the better.

Performance: Figure 1 shows the normalized performance with SpecPref compared to the insecure baseline that performs prefetching even during speculative execution. On average, SpecPref provides no performance slowdown with IPCP, SPP and PPF. For IPCP, the performance with SpecPref improves even above the baseline by 0.62%. For FNL+MMA, the average slowdown reduces to less than 1%. However, for Bingo, there is still a slowdown of 1.17%. Since Bingo prefetches based on a memory region; thus, prefetching only one block, speculatively, does not improve its performance significantly. The maximum slowdown across all benchmarks and all the prefetchers has reduced from 20.46% (for SPP) to 14.53% (for IPCP). We find that benchmarks like `gcc` and `fotonik3d` perform even better than the insecure baseline. This is due to the reduced latency for accessing the prefetched data in L0 and SPC.

The SpecPref implementation outperforms Ghost Loads due to the distribution of prefetching requests amongst the speculative and the non-speculative caches. For Ghost loads at L1D, the performance slowdown increases due to complete speculative prefetching that induces thrashing with IPCP. The conservative prefetching mechanism of SpecPref avoids the possibility of thrashing in the small speculative cache.

Prefetch coverage: The performance improvement with SpecPref is attributed to the decrease in prefetch lateness, which improves the prefetch coverage. For SpecPref, when compared to MuonTrap with on-commit prefetching, in terms of prefetch coverage, FNL+MMA, IPCP, Bingo, SPP, and PPF improve average (maximum) coverage by 11.34% (23.58%), 4.79% (34.44%), 7.54% (34.59%), 5.17% (34.61%), and 4.65% (36.09%), respectively.

Speculative cache size sensitivity: To study the effect of speculative cache sizes on the performance of hardware prefetchers, we vary the cache sizes for L0D, L0I, and SPC from 2KB to 32KB (Figure 5). We evaluate the hit latencies by using PCACTI [18], thus we consider a 1-cycle hit latency for the 2KB and 4 KB caches. For 8KB, 16KB, and 32KB caches, we consider a hit latency of two, three, and four cycles, respectively. For L0I and L0D, when we increase the cache size from 2KB to 4KB, the performance with SpecPref increases due to larger cache size with equivalent hit latency. However, with further increase in the cache size, the performance improvement goes down drastically as the increased hit latency becomes a bottleneck. For ghost prefetching at L1D, the performance improves with an increase in the cache size (except 32KB), this is due to the reduced thrashing as the capacity of the cache increases.

TABLE II: Parameters of the simulated system.

Core	One or eight cores, 4 GHz, 6-issue, 352-entry ROB, 12-entry IQ, 128-entry LQ, 72-entry SQ
L0-D and I Cache	2KB, 4-way, 1-cycle, PQ:16, 16 MSHRs
L1-D Cache	48KB, 12-way, 5-cycle, PQ:16, 16 MSHRs, Prefetcher: IPCP
L1-I Cache	32KB, 8-way, 4-cycle, PQ:16, 8 MSHRs, Prefetcher: FNL+MMA
L2 Cache	512KB, 8-way, 10-cycle, PQ:16, 32 MSHRs, Prefetcher: Bingo, SPP, or PPF
LLC	2 MB/core, 16-way, 20-cycle, PQ:32x#cores, MSHR: 64x#cores
Cache line size	64B in L0, L1, L2 and LLC
Memory	1 channel for single-core, 2 channels/multi-core, 8 banks/rank, 3200 MT/sec, 64 read-/write queues, FR-FCFS

Since the SPC has a latency lower than L2, the maximum latency to access the L2 always remains the baseline L2 latency. Thus with an increase in the SPC size, the latency to access the L2 level does not increase, but instead the increase in cache size improves performance. Bingo performs better with large SPC sizes since more speculative prefetch requests can be invoked without thrashing the SPC, which improves the timeliness and in turn prefetch coverage. Ghost Loads approach performs better with an SPC of size above 16KB since a large cache easily accommodates all the speculatively prefetched blocks without getting thrashed.

On-chip bandwidth demand: Figure 6 shows the normalized bandwidth with MuonTrap, Ghost Loads and SpecPref compared to an insecure baseline. Note that the Ghost Loads technique does not consider the instruction prefetchers. We evaluate the bandwidth using the L0 speculative cache size of 2KB with FNL+MMA and IPCP, with the L2C prefetchers, we consider an SPC of size 16KB. On average, SpecPref demands an additional bandwidth of 24.38%, 3.61%, 26.16%,

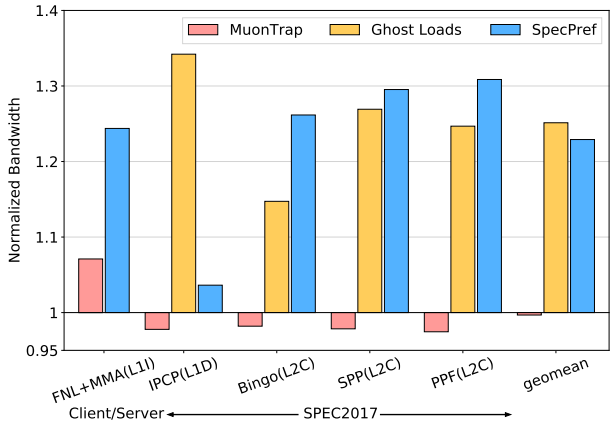


Fig. 6: Normalized bandwidth usage with the three techniques compared to an insecure baseline. Lower the better.

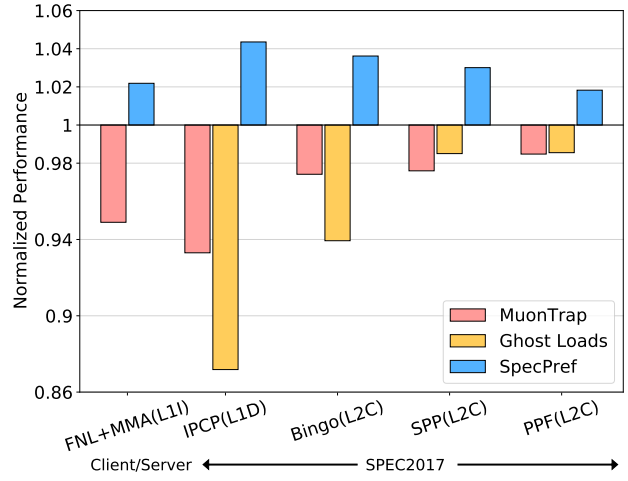


Fig. 7: Normalized 8-core performance compared to an insecure baseline.

29.58%, and 30.93% for FNL+MMA, IPCP, Bingo, SPP, and PPF, respectively. For FNL+MMA and IPCP, we report L1-L2 bandwidth, and for the rest, we report L2-LLC bandwidth. There is a marginal increase (less than 5%) in bandwidth demand between L0 and L1, with the exception of FNL+MMA.

Ghost Loads requires an additional bandwidth of 34.21%, 14.73%, 26.92% and 24.68% above the insecure baseline for IPCP, Bingo, SPP, and PPF respectively. The excessive bandwidth increase for IPCP is attributed to the increase in the number of misses due to extensive thrashing at L0D.

Multicore results: Figure 7 shows the multicore performance with MuonTrap, Ghost Loads and SpecPref, simulated on an 8-core system with 100 multi-programmed homogeneous and heterogeneous mixes, similar to [9], [11], [12]. On average, MuonTrap shows a performance slowdown of 5.10%, 6.70%, 2.58%, 2.40% and 1.52% for FNL+MMA, IPCP, Bingo, SPP, and PPF, respectively. Ghost Loads shows a performance slowdown of 12.82%, 6.07%, 1.50% and 1.45% with IPCP, Bingo, SPP and PPF respectively. SpecPref improves the performance above the baseline by 2.19%, 4.35%, 3.62%, 3.01%, and 1.83% for the previously stated prefetchers respectively. Compared to single-core, Specpref's effectiveness increases with multicore as timely prefetching and higher coverage has more utility in multicore system as it helps in reducing shared resource contentions at the L3 and DRAM.

V. Conclusion

This paper presented SpecPref, an enhancement that improves the performance of hardware prefetchers with secure cache hierarchies that are resilient to speculative execution attacks. SpecPref considers the state-of-the-art instruction and data prefetchers for evaluation as opposed to Ghost Loads. Our discussion showed that the increase in the prefetch lateness can reduce the prefetch coverage, consequently reducing the performance gain provided by the prefetchers. SpecPref demonstrated that it is possible to balance the prefetching load between speculative and on-commit execution to improve performance while maintaining security. On average, SpecPref reduces the performance slowdown for the five evaluated prefetchers from as low as no slowdown to an average of 1.17% slowdown, with marginal changes to the cache hierarchy. In conclusion, We have shown that it possible to speculatively prefetch cache blocks and not to delay the prefetching until the commit time while still providing the security.

References

[1] https://research.ece.ncsu.edu/ipc/infrastructure/
[2] https://dpc3.compas.cs.stonybrook.edu/?SW_IS
[3] Ainsworth and Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in ISCA 2020, pp. 132-144
[4] https://en.wikipedia.org/wiki/Sunny_Cove_(microarchitecture)
[5] https://dpc3.compas.cs.stonybrook.edu/
[6] A. Ros and A. Jimborean, "The entangling instruction prefetcher," IEEE Computer Architecture Letters, vol. 19, pp. 84-87, 2020.
[7] A. Seznec, "The FNL+ MMA Instruction Cache Prefetcher," First Instruction Prefetching Championship, 2020.
[8] Gupta et al., "Seclusive Cache Hierarchy for Mitigating Cross-Core Cache and Coherence Directory Attacks," in DATE 2021, pp:1-4
[9] Pakalapati and Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in ISCA 2020, pp. 118-131
[10] Bakhshalipour et al., "Bingo spatial data prefetcher," in HPCA 2019, pp. 399-411
[11] Kim et al., "Path confidence based lookahead prefetching," in MICRO 2016, pp. 1-12
[12] Bhatia et al., "Perceptron-based prefetch filtering," in ISCA 2019, pp. 1-13
[13] Kocher et al., "Spectre attacks: Exploiting speculative execution," in S&P 2019, pp. 1-19
[14] A. Seznec, "A 256 kbits l-tage branch predictor," Journal of Instruction-Level Parallelism (JILP) 2007, vol. 9, pp. 1-6
[15] "Champsim simulator." [Online]. Available: https://github.com/ChampSim/ChampSim
[16] "Spec cpu 2017." [Online]. Available: https://www.spec.org/cpu2017/
[17] The 1st Instruction Prefetching Championship. [Online]. Available: https://research.ece.ncsu.edu/ipc/
[18] Pcacti tool. [Online]. Available: https://sportlab.usc.edu/downloads/download/
[19] Koruyeh et al., "Spectre Returns! speculation attacks using the return stack buffer," in WOOT 2018.
[20] Maisuradze and Rossow, "Ret2spec: Speculative execution using return stack buffers," in SIGSAC 2018, pp. 2109-2122
[21] Schwarz et al., "NetSpectre: read arbitrary memory over network," in ESORICS 2019, pp. 279-299
[22] Bhattacharyya et al., "SMoTherSpectre: exploiting speculative execution through port contention", in CCS 2019, pp. 785-800

[23] Yan et al., "Invisispec: Making speculative execution invisible in the cache hierarchy," in MICRO 2018, pp. 428-441
[24] Saileshwar and Qureshi, "Cleanupspec: An "undo" approach to safe speculation," In MICRO 2019, pp. 73-86
[25] Zhao et al., "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in HPCA 2019, pp. 264-276
[26] Khasawneh et al., "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in DAC 2019, pp. 1-6
[27] Fustos et al., "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," In DAC 2019, pp. 1-6
[28] Sakalis et al., "Ghost loads: What is the cost of invisible speculation?," In CF 2019, pp. 153-163
[29] Behnia et al. "Speculative interference attacks: Breaking invisible speculation schemes," In ASPLOS 2021, pp. 1046-1060