IOS: A Low Cost Defense to Mitigate Meltdown and Spectre like Attacks

Xin Wang

Department of Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA 23284
wangx44@vcu.edu

Wei Zhang

Department of Computer Science and Engineering
University of Louisville
Louisville, KY 40292
wei.zhang@louisville.edu

***Abstract*—The Meltdown and Spectre attacks brings severe security issues to a wide range of modern processors. The Meltdown steals sensitive information in the kernel memory space by utilizing the nature of out-of-order execution and the Spectre exploits the speculative execution to access the secret data. Both attacks transmit the secrets via the cache side-channels. Although software patches have been applied to protect the processors from the Meltdown and Spectre attacks, the countermeasure also introduces huge performance degradation and close the door to the benefit of out-of-order and speculative execution. Customizing a hardware-based solution can be more performance friendly and also conserve the bonus the out-of-order and speculative execution. In this paper, we proposed a hardware-based mitigation technique named Invalidation on Squash (IOS) which is able to close the cache covert channel and stops Meltdown and Spectre from exposing the secrets to the adversary. To simplify the additional defense logic and minimize the hardware overhead, IOS targets the squashed load instructions and invalidates the corresponding cache lines introduced by these squashed loads. Compared to the existing Meltdown and Spectre Countermeasures, IOS incurs only negligible hardware overhead by taking the advantage of the simple invalidation logic.**

***Index Terms*—Security, Cache Covert Channel, Meltdown, Spectre, Countermeasure**

I. Introduction

Out of Order and Speculative execution have been widely leveraged by modern processors to pursue performance enhancement and instruction level parallelism exploitation. However, Out of Order and Speculative execution also expose fundamental security vulnerabilities which are uncovered by the recent Meltdown [1] and Spectre [2]. Consequently, the adversaries are able to the access kernel memory space and steal secrets by taking advantage of the vulnerabilities offered by the strategies that were intended to promote the performance.

Both Meltdown and Spectre follow a two-steps style to retrieve the secret. The first step is to utilize transient instructions to expose the secret information. For Meltdown, a transient instruction is the instruction executing in an OoO (Out of Order) way. For Spectre, a transient instruction is the one that is unexpectedly executed due to a mis-prediction. Since a transient instruction is always unexpected, it is then able to be executed with violations (e.g. accessing the unauthorized memory). Although the control flow of the execution pipeline can flush the intermediate states of the transient instruction, the clean-up is only effective inside the pipeline and other state changes outside of the pipeline can be conserved, e.g. the data cache may still hold the secret relevant data. In the second step, the secret information is transmitted to the adversary via a covert channel. The cache-based covert timing channel is the efficient and robust and broadly used by attackers. Among different cache attack techniques, Flush+Reload [3] is the most famous and has been used for attacks on varieties of computation algorithms.

In this paper, we proposed a hardware-based countermeasure named IOS that can clean up the cache status of the transient instructions.The transient instruction will be eventually squashed due to the violation (e.g. cross the memory boundary) and the intermediate status within the execution pipeline will be uninstalled. Additionally, IOS enables the execution pipeline to evict also the status of the the memory hierarchies including L1, L2 cache and etc. The IOS removes the secret information brought up by the transient instruction and cuts off the cache covert channel connecting to the adversary. The previous hardware-based mitigation solutions rely heavily on complex logic, new micro-architecture and additional memory space to resolve the threats of Meltdown and Spectre at a low cost of the performance penalty. These solutions, however, increase the implementation complexity and hardware overhead. Instead of pursuing low performance degradation, the proposed IOS implements a lightweight strategy that invalidates the cache lines installed by transient load instructions in a very aggressive way. The experimental results show that the performance loss (7.61%) is still low enough as compared to one of the state-of-art Meltdown&Spectre countermeasures that achieves the lowest performance degradation (5.1% in [4]). And due to its simplicity, the IOS can be supported by no additional micro-architecture and negligible space and leads to only negligible hardware overhead.

II. Background and Related Work

A. Meltdown&Spectre-like Threats

To fill the stalls from the data and control dependencies and achieve peak performance, the modern processors execute instructions in a speculative out-of-order way. However, the side-effect of the speculation and out-of-order execution is the transient instruction which has been demonstrated to be a vulnerability of exposing critical information to the adversary. The most famous attacks that rely on the transient instruction to leak secrets are Meltdown [1] and Spectre [2]. Both attacks leverage the transient instructions to access the privilege memory space and dump the secret through the cache side-channels. The transient instruction in the Meltdown attack is an instruction with violations and it is not on the execution path. This transient instruction should never be executed. However, due to the processors' out-of-order execution ability, the transient instruction is executed and install secret relevant information to the CPU's micro-architectures (e.g. CPU's memory hierarchies including L1/L2 caches). Although it will eventually be discarded as it leads to violations, the state change in micro-architectures is preserved.

In Spectre, the mis-prediction is intentionally trained to direct the program to execute in a wrong path toward the transient instruction. Although the transient instruction will be discarded after the prediction has been resolved and the execution has been corrected to the right path, similar to that in Meltdown, the secret relevant change of state in cache hierarchies has survived. By using cache side-channels, the adversary is able to probe the transient state in the cache hierarchies including L1 data cache, L2 cache and last level cache, recover the secret and transmit it to the outside. Flush+Reload [3], Prime+Probe [5] and Evict+Time [5] are cache side-channel attacks demonstrated and Flush+Reload is the most widely used cache channel attacks owing to its high bandwidth and robustness. The timing-based cache side channel can also be defended with constant time operations [6], [7].

As show in Figure 1, the Meltdown attack leverages a transient instruction to access an entry of an array. The size of the entry can fit to a cache line and the entry index is computed using the secret value. As the result of executing the transient instruction, a specific cache line indexed with secret value is loaded to the cache and it is preserved after the transient instruction has been squashed. The attacker reads each cache line and reveal secret value by observing a hit to the specific cache line previously installed by the transient instruction.

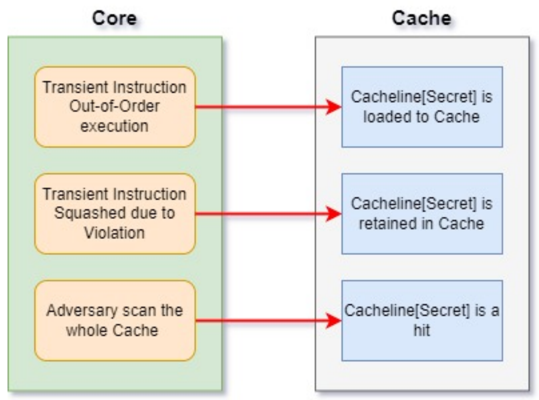
Core
Cache
Transient Instruction Out-of-Order execution
Cacheline[Secret] is loaded to Cache
Transient Instruction Squashed due to Violation
Cacheline[Secret] is retained in Cache
Adversary scan the whole Cache
Cacheline[Secret] is a hit

Fig. 1. The Meltdown attack leaks secrets via Flush+Reload cache side-channel.

B. Existing Mitigation

Several hardware-based countermeasures against Meltdown&Spectre-like attacks have been proposed. In order to prevent a transient instruction from changing the cache state, DAWG [8] proposed to add protection domain to cache hierarchies. Invisispec [9] proposed to involve shadow structures to avoid the side-effects of caches from the speculative execution. Conditional Speculation proposed in [10] blocks the cache requests until the branch prediction has been resolved and the correct path has been determined. The transient instruction then is eliminated and no secret will be revealed by the Spectre attack. NDA—Non-speculative Data Access, proposed in [11] proposed a technique to restrict speculative data propagation in out-of-order (OoO) processors. CleanupSpec [4] provides a similar strategy with the IOS to ”undo” the state changes installed by a transient instruction, but it is composed of much more complicated logic and needs relatively more hardware resources to accomplish the defense against Meltdown and Spectre. CleanupSpec undoes the changes to the cache hierarchies caused by transient instructions and it achieves very low performance overhead by intending to recover several kinds of changes besides squashing the cache lines loaded by the transient instruction. Although these hardware-based countermeasure can effectively mitigate the Meltdown&Spectre-like attacks and achieve much lower performance degradation than software-based solutions, they require additional hardware resources and non-trivial logic to support the mechanism and consequently involves hardware overhead that increasing the area and power consumption of the processors. As comparison, the IOS proposed in this paper focuses on squashing the dangerous cache lines installed by the transient instructions and makes the cleanup as deeper as possible to close the cache side-channel and thus prevents the processors from the threats of Meltdown&Specre attacks. Moreover, the IOS tailors to use arbitrary cleanup strategy which is unified for all the cache hierarchies to achieve negligible hardware overhead.

III. IOS: A Lightweight Defense

The IOS intends to achieves to protect modern processors from the threats of Meltdown&Spectre-like attacks, and also minimize the hardware overhead and conserve the performance profit of OoO and Speculative execution as much as possible. The IOS proposes a lightweight strategy to thoroughly purify the state changes of the cache subsystem.

Thanks to the arbitrariness, the IOS can build the cleanup process with straight-forward logic and very few hardware resource. As shown in Figure 2, the change of states in all cache levels are erased by the IOS, therefore, the footprint of the transient load instruction is purely cleaned up after it is squashed in the execution pipeline.

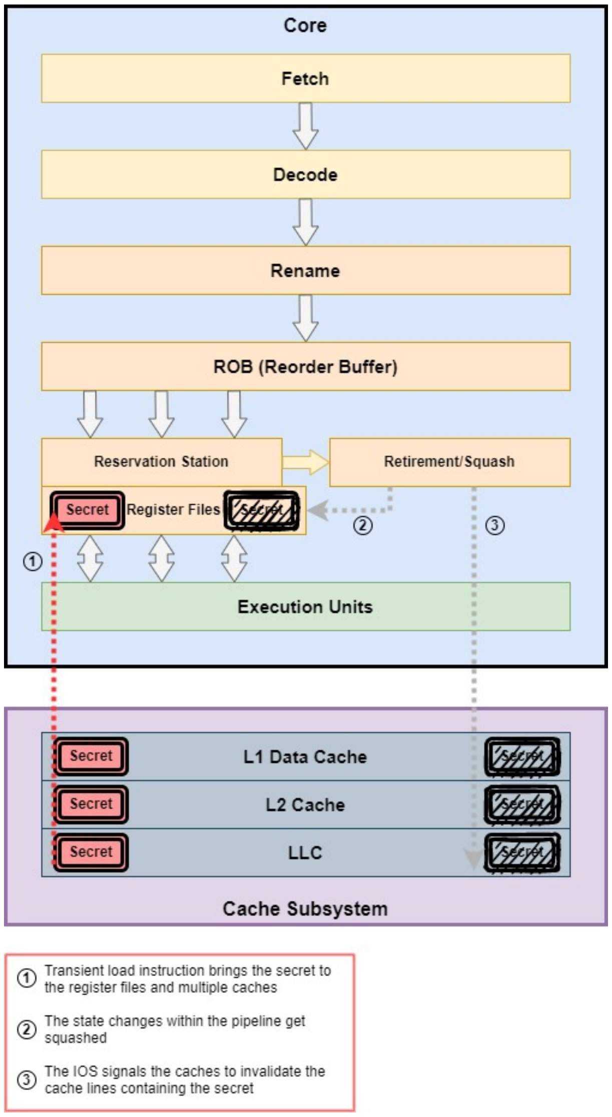
Core
Fetch
Decode
Rename
ROB (Reorder Buffer)
Reservation Station
Retirement/Squash
Secret
Register Files
Secret
①
②
③
Execution Units
Secret
L1 Data Cache
Secret
Secret
L2 Cache
Secret
Secret
LLC
Secret
Cache Subsystem
① Transient load instruction brings the secret to the register files and multiple caches
② The state changes within the pipeline get squashed
③ The IOS signals the caches to invalidate the cache lines containing the secret

Fig. 2. The IOS cleans up the state changes in caches.

Target only transient load instructions. The IOS only applies the cleanup process to the transient load instructions. These instructions are able to bring secret related data to the caches. Therefore the performance bonus from OoO and Speculative execution can still be guaranteed for the applications that contains less transient load instructions.

Send invalidation signals to caches on squash. The IOS arbitrarily sends invalidation signal to the first level cache when a transient load instruction is being squashed and the invalidation signal doesn't expect a response from the cache. Since there is not payload for the invalidation signal and it is unnecessary for the caches to reply the sender, the IOS has only slight impact on the bus traffic. Due to the arbitrary invalidation strategy, there is no complex logic needed to evaluate the invalidation condition and no additional memory space are required to maintain any instruction or cache status.

Pass invalidation signal to all cache levels. To erase the state changes of caches as deeply as possible, the IOS intends to spread the invalidation signal to all the cache levels. As shown in Figure 3, L1 data cache receives the invalidation signal and kick out the secret. The L1 data cache then wrap up an invalidation signal and pass it on to lower level cache which is the L2 cache. The L2 cache removes the secret after the invalidation signal arrives. If the LLC exists, the L2 cache should notify LLC to do the same cleanup by handing over the invalidation signal to the LLC. Regardless of how many cache layers in the processor, the invalidation signal will always reach to the bottom of the cache subsystem to perform a thorough cleanup toward the secret.

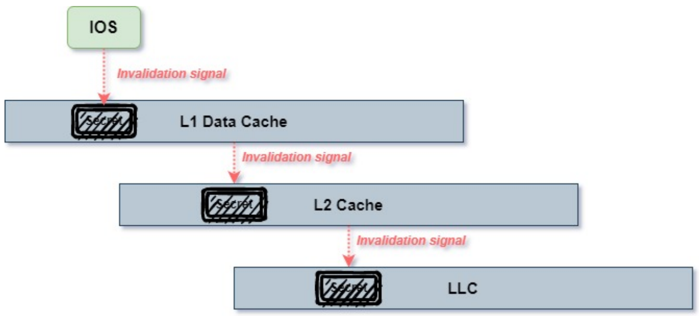

Fig. 3. Erase State Changes in Caches.

Invalidate if hit, skip copying if miss. The invalidation signal is similar with a load request to the cache. If the secret has already be loaded to the cache before the transient load instruction is squashed, the invalidation signal can hit the cache line that contains the secret and will simply remove the secret from the cache. On the other hand, at the point of squashing a transient load instruction, the missing data requested by the transient load may not be filled to the cache yet. Since the data has not arrived, the invalidation signal cannot find a valid cache line to evict and the ISO consider this case as a cache miss of the invalidation signal. Instead of invalidating the secret, a fake cache line is inserted to the cache and the status of this cache line is set to **locked**. The tag of the fake cache line is the same with the cache line where the secret is about to be added to. When the data returns from the lower level of the memory hierarchy, a free cache line will be picked up to accommodate the data. Instead, the fake cache line with **locked** status is returned and advices the cache to skip copying the data. Therefore, the state of the cache is not modified by the transient load instruction. The fake cache line is then released as well. To summarize, the IOS 1) kicks the secret out of the cache if the invalidation signal has a cache hit; 2) utilizes a fake cache line to prevent the upcoming secret being copied to the cache if the invalidation signal has a cache miss.

IV. Methodology & Experimental Results

We used Gem5 [12] to evaluate the IOS. Gem5 is a cycle accurate simulator and it is able to model the out-of-order processor. We configure the Gem5 to System-call Emulation (SE) mode to simulate a single-core processor. The processor configuration including cache hierarchy and execution engine parameters is shown in Table I. 17 benchmarks from SPEC-CPU2006 [13] are used to evaluate our IOS. SPEC-CPU2006 is a set of benchmarks designed to evaluate the performance of modern processors. For each benchmark, the first 500 Million instructions are warm-up instructions and the following 100 Million instructions are observation instructions.

TABLE I
Simulated CPU architecture configuration

Processor	Single OoO core, 2GHz
L1 DCache	32KB, 8-Way, 64B cache line
L1 ICache	32KB, 8-Way, 64B cache line
L2 Cache	2MB, 16-way, 64B cache line
Execution Engine	192 ROB Entries, 32 LSQ Entries 32 IQ Entries, 4096 BTB Entries

A. Performance Degradation

The IOS results in 8.27% performance loss which is good enough as compared to the countermeasure proposed in [4] that limits the execution slowdown to around 5%. We evaluated 17 benchmarks from SPEC2006. The experimental results of these benchmark show that, 1) 7 out of 17 benchmarks has over 5% performance degradation; 2) other 10 benchmarks only suffer from the performance loss that is below 5%. As shown in Figure 4, serious performance side-effect can be found in the benchmarks such as **464.h264ref**, **473.astar**, **471.omnetpp** and **400.perlbench**. For these benchmarks, the IOS incurs around 20% execution slowdown. On the other hand, **459.GemsFDTD**, **435.gromacs**, **470.lbm**, **434.zeusmp** and **429.mcf** are typical benchmarks that are IOS friendly and have only negligible performance loss (less than 1%) or even slight performance improvement.

B. Workload Profile

As mentioned in IV-A, the IOS may hurt the performance more badly for the applications that have more load instructions that are squashed during the execution. As shown in Figure 5, 7 of 17 benchmarks experience more than 5% execution slowdown and 5 of them have more amount of squashed loads than the average except **400.perlbench** and **465.tonto**. For the other 10 benchmarks that are less affected by the IOS, are all having the amount of the squashed loads below the average except **459.GemsFDTD**.

Moreover, by comparing the baseline to the IOS for the squashed loads amount, the benchmarks with significant performance loss (more than 5%) shows sharply increase of the squashed loads after the IOS is deployed and the benchmarks with insignificant performance loss (less than 5%) shows mild increase or even slight decrease of the squashed loads after applying the IOS. The scatter line shown in Figure 5 depicts that the incremental rate is up to 36.8% for the benchmarks with more than 5% performance penalty and the incremental rate is less than 5.3% for the benchmarks with slight performance loss. The outlier is benchmark **462.libquantum**. The number of the squashed loads increases 10.7% but the performance degradation only is 3.9% for **462.libquantum**.

V. Conclusion

In this paper, we proposed a hardware-based countermeasure called Invalidation on Squash (IOS) to protect the modern processors against the the Meltdow&Spectre-like attacks, which are examples of microarchitectural vulnerabilities [14]. The IOS is able to stops Meltdown and Spectre from exposing the secrets to the adversary. IOS targets all the squashed load instructions and invalidates the corresponding cache lines introduced by these squashed loads. Taking advantage of the arbitrary invalidation logic, IOS incurs only negligible hardware overhead compared to the existing Meltdown and Spectre countermeasures. According to the experimental results, the IOS can prevent the Meltdow&Spectre-like attacks at a cost of 8.27% overall performance degradation which is still competitive with other hardware-based countermeasures. In our future work, we aim to explore methods for enhancing the IOS approach by expanding protect instructions intelligently without incurring significant performance overhead or hardware cost.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[3] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[4] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[6] "Intel corporation. 2019. guidelines for mitigating timing side channels against cryptographic implementations," 2019.

[7] I. Puddu, M. Schneider, M. Haller, and S. Čapkun., "Frontal attack: Leaking control-flow in sgx via the cpu frontend," in *USENIX Security*. IEEE, 2021, pp. 663—680.

[8] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[9] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[10] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 264–276.

[11] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Micro*. IEEE, 2019, pp. 572–586.

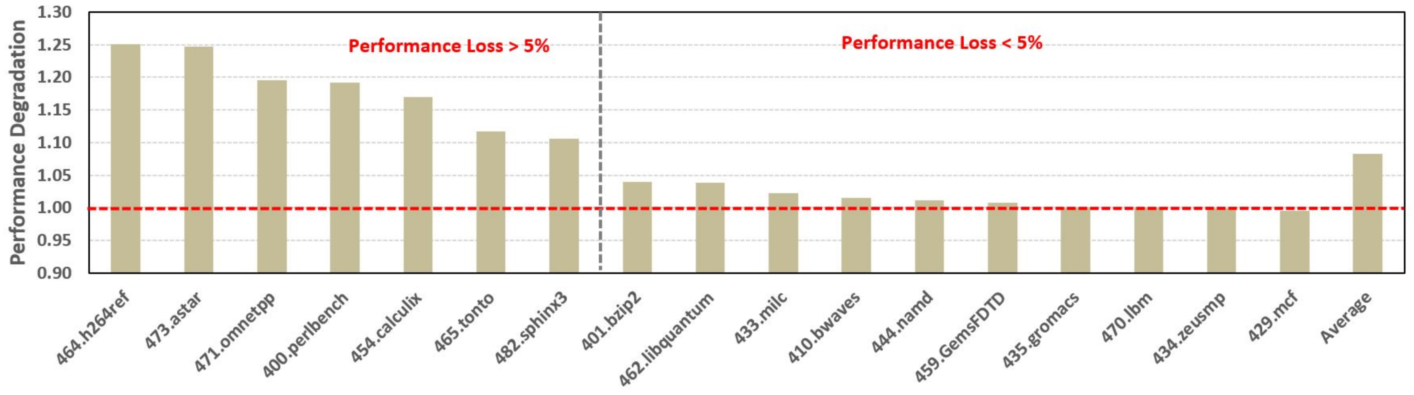

Fig. 4. Execution cycles of the IOS normalized to the baseline w/o IOS Protection.

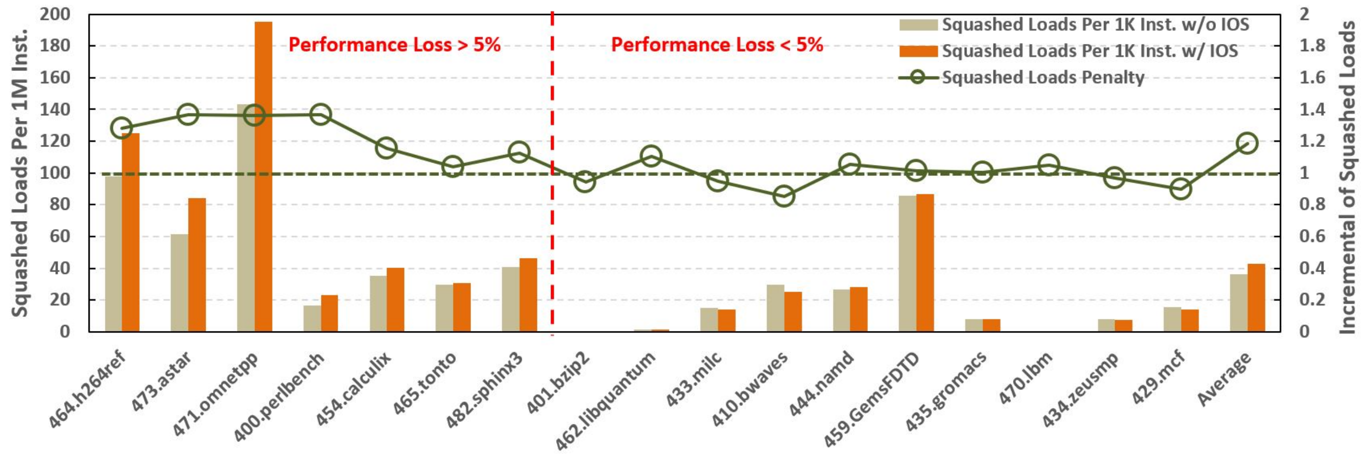

Fig. 5. Number of Loads Squashed Per 1K Inst.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[13] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[14] J. Zhang, C. Chen, J. Cui, and K. Li, "Timing side-channel attacks and countermeasures in cpu microarchitectures," in *ACM Computing Surveys*. ACM, 2024, pp. 1—40.