Mitigation of Hardware Attacks

Research Project Report

Submitted in partial fulfillment of the requirements
for the completion of

EE 691

by

Rishabh

(Roll No. 200260041)

Under the guidance of

Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
October 2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

Rishabh
Electrical Engineering
IIT Bombay

Abstract

The advent of multi-core systems saw the introduction of shared Last Level Caches between cores to improve performance. Different users working on different cores of the same processor accessed the same LLC. The attacker could probe the LLC and identify accesses by the victim based on access times, which could lead to the leak of sensitive information.

Attacks like Prime+Probe[1] and Flush+Reload[2] exploit the newly formed side channel to breach the victim's privacy and gather sensitive information. They monitor the access times for different lines in the LLC; a more significant access time implies it was present in the DRAM and the victim did not access it, whereas a shorter access time reveals access.

PASSP[3] is an idea developed to mitigate these vulnerabilities of a shared cache. It invalidates those blocks that are transferred from one core to another.

The project aims to develop the PASSP[3] further by incorporating better replacement algorithms and modifying the partitioning method.

Contents

List of Figures

Chapter 1

Introduction

The cache memory is a resource that does not need to be explicitly managed by the user. Instead, a set of replacement policies (called cache algorithms) handles cache management. These determine which data is to be stored in the cache during the execution of a program. To be both costs- effective and efficient, caches are usually several orders of magnitude smaller than main memory (e.g., there are typically a few KB of L1-cache and a few MB of L3-cache versus many GB or even a few TB of main memory). Consequently, the dataset we are currently working on (the working set) can easily exceed the cache capacity for many applications. Handling this limitation involves cache algorithms that address the questions of

1. Which data do we load from main memory, and wherein the cache do we store it?
2. If the cache stands already full, which data do we evict?

If the CPU requests a data item during program execution, it first determines whether it exists in the cache. If this is the case, the request can be serviced by reading from the cache without requiring a time-consuming main memory transfer. This is referred to as a cache hit. Otherwise, we have a cache miss. Cache algorithms aim at optimizing the hit ratio, i.e., the percentage of data requests resulting in a cache hit.

In multi-core systems, the first-level and, in most cases, the second-level caches are private to each core, whereas a shared third-level cache exists between the cores. The LLC remains partitioned to allocate some lines for each core. Static Partitioning divides it equally between each core once and never again. This, in principle, is similar to a private LLC. Whereas UCP dynamically allocates lines to cores based on their utility or demand for lines. A core with greater demand than another is given an extra line taken from one of the lines previously allocated to the other core.

An adequate replacement policy could help better select the line to be reallocated, and a set-wise UCP could also help better partition sets.

Chapter 2

Literature Survey

2.1 UCP

Utility Based Partitioning is a runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources.

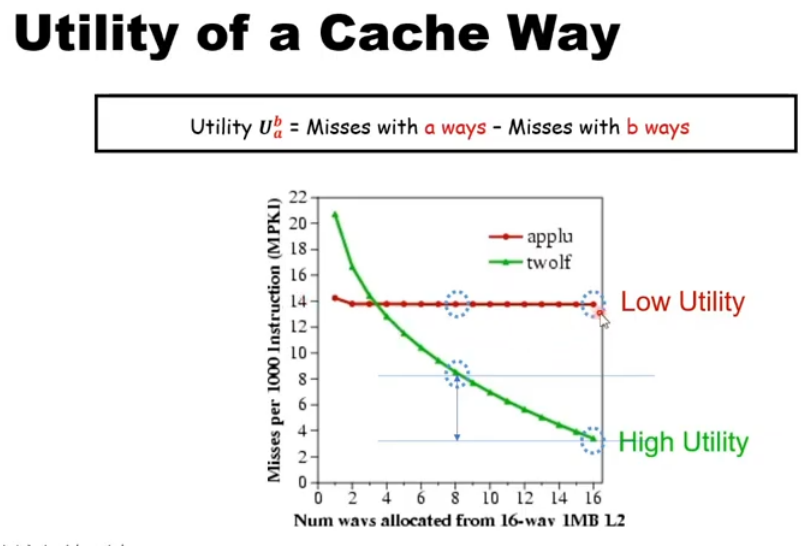

Figure 2.1: Utility of a cache

A replacement policy implicitly partitions a shared cache on a demand basis, giving more cache resources to the application that has a high demand and fewer cache resources to the applica- tion that has a low demand. However, a higher demand for cache resources does not always correlate with a higher performance from additional cache resources. It is beneficial for performance to invest cache resources in the application that benefits more from the cache resources rather than in the application that has more demand for the cache resources

2.2 Probe + Prime

PRIME +PROBE [1], is a general technique for an attacker to learn which cache set is accessed by the victim core. The attacker, runs a spy process which monitors cache usage of the victim, as follows: The attacker fills one or more cache sets with its own code or data. This is the PRIME

phase. It then waits for a pre-configured time interval while the victim executes and utilizes the cache. This is the IDLE phase Finally the attacker continues execution and measures the time to load each set of his data or code that he primed. If the victim had accessed some cache sets, it will have evicted some of the attacker's lines, which it observes as increased memory access latency for those lines. This is the PROBE phase.

Figure 2.2: Prime + Probe Algorithm

Attacker takes lines and primes them. The attacker then returns these lines. The victim program executes.Finally the attacker takes these lines back and probes addresses A1 through A6

2.3 Flush + Reload

Page sharing exposes processes to information leaks. FLUSH +RELOAD [2], a cache side-channel attack technique that exploits this weakness to monitor access to memory lines in shared pages. A round of attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The spy, then, waits to allow the victim time to access the memory line before the third phase. In the third phase, the spy reloads the memory line, measuring the time to load it. If during the wait phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take signifi- cantly longer.

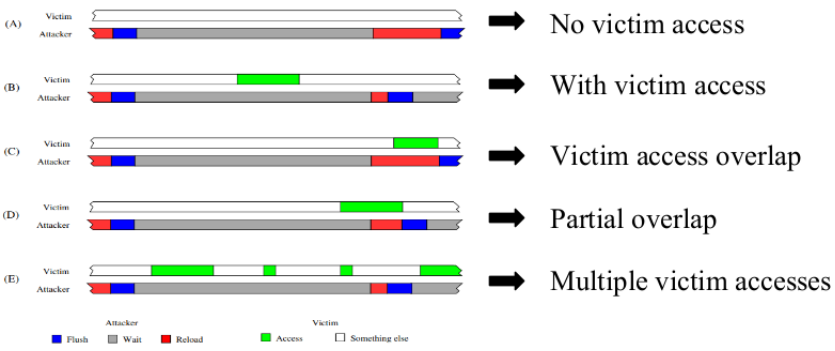

Figure 2.3: Flush + Reload Algorithm

2.4 PASS-P

Performance And Security Sensitive Dynamic Cache Partitioning [3] deals with the kind of attacks in which the at- tacker tries to analyze the memory accesses made by the victim to find out which parts of the victim pro- gram have been executed such as Flush+Reload[2] and Prime+Probe [1]. In order mitigate these, the attacker must be prevented from successfully performing differential timing analysis on the reallocated lines. To stop the access of shared resources of other program, PASS-P invalidates all cache lines that are reallocated from one process to another. Because of this preemptive invalidation of lines, no process is able to cause eviction of lines of any other process.

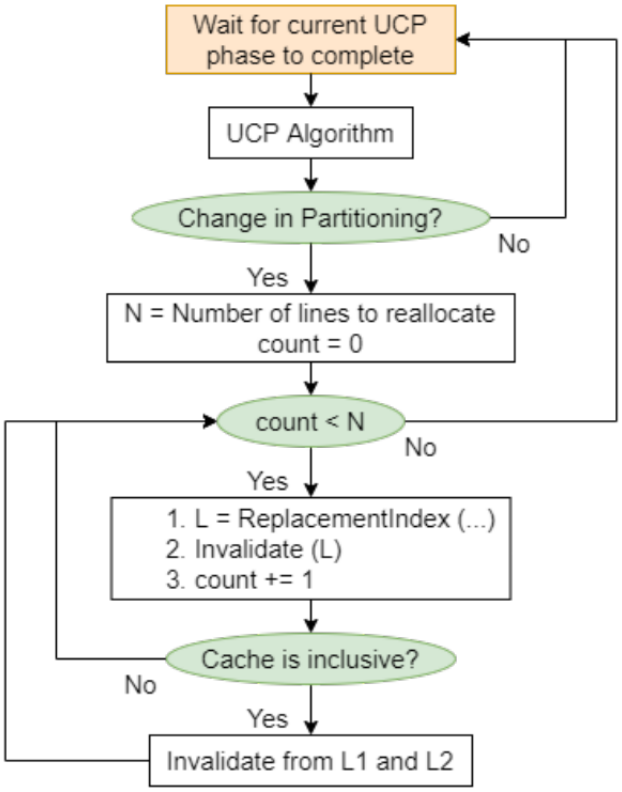

Figure 2.4: Flow graph depicting the PASS-P algorithm

2.5 DAAIP

Deadblock Aware Adaptive Insertion Policy (DAAIP) [4] which dynamically adapts to the chang- ing cache behaviour of applications sharing the LLC by changing the insertion policy. A counter is maintained to count the number of deadblocks encountered, and after a large number of cycles it is compared against a threshold. If it is greater the insertion policy is changed to insert at LRU whereas if it is less than the threshold, the insertion policy inserts at LRU - 1.

Chapter 3

Proposed Idea

3.1 UCP with Invalidation and DAAIP

PASS-P [3] overcomes the shortcomings of having a shared cache that compromises performance. The invalidation prevents the other core from comparing its tag during access against the newly reallocated line. It decreases the hit rates and, indirectly, the performance. PASS-P's performance could increase if a more effective algorithm were used to select lines to be reallocated and invalidated.

DAAIP [4] is an effective replacement algorithm, and it would facilitate the reallocation of the dead- blocks between cores. DAAIP performs as a better cache replacement algorithm than SRRIP or LRU could be used to increase the performance of utility-based cache partitioners.

Counters are present to count the number of hits and misses the LLC receives. This finds the total number of accesses, the miss rate, and the cache's utility. This also computes the utility as the difference in the number of misses for each core in two successive ways. Ways here refers to the number of lines owned by a core due to partitioning. The core with a greater utility and hence a higher miss rate demands more lines.

Each set gets partitioned identically, but the line owned by the cores in each set differs. An algorithm identifies an eviction candidate for each set and core. Moreover, when partitioned, its ownership is transferred to the other core. Furthermore, the partitioning is done only on a cache miss. Once partitioned for the cache, each set is partitioned only upon a cache miss. The old partitioning persists if it keeps receiving hits.

Additionally, the partitioning is done once every 10,000 cache accesses to support this idea. Another feature that could improve performance would be partitioning each cache set based on its own utility. Each cache set would maintain the number of accesses and hits it receives and compute its utility for the current and previous partitions. The difference would then revert or further partition the set to meet demands. The DAAIP replacement algorithm would select the lines to be reallocated. The partition is done once every 1000 set accesses. This would allow the partition to adapt faster to the core demands.

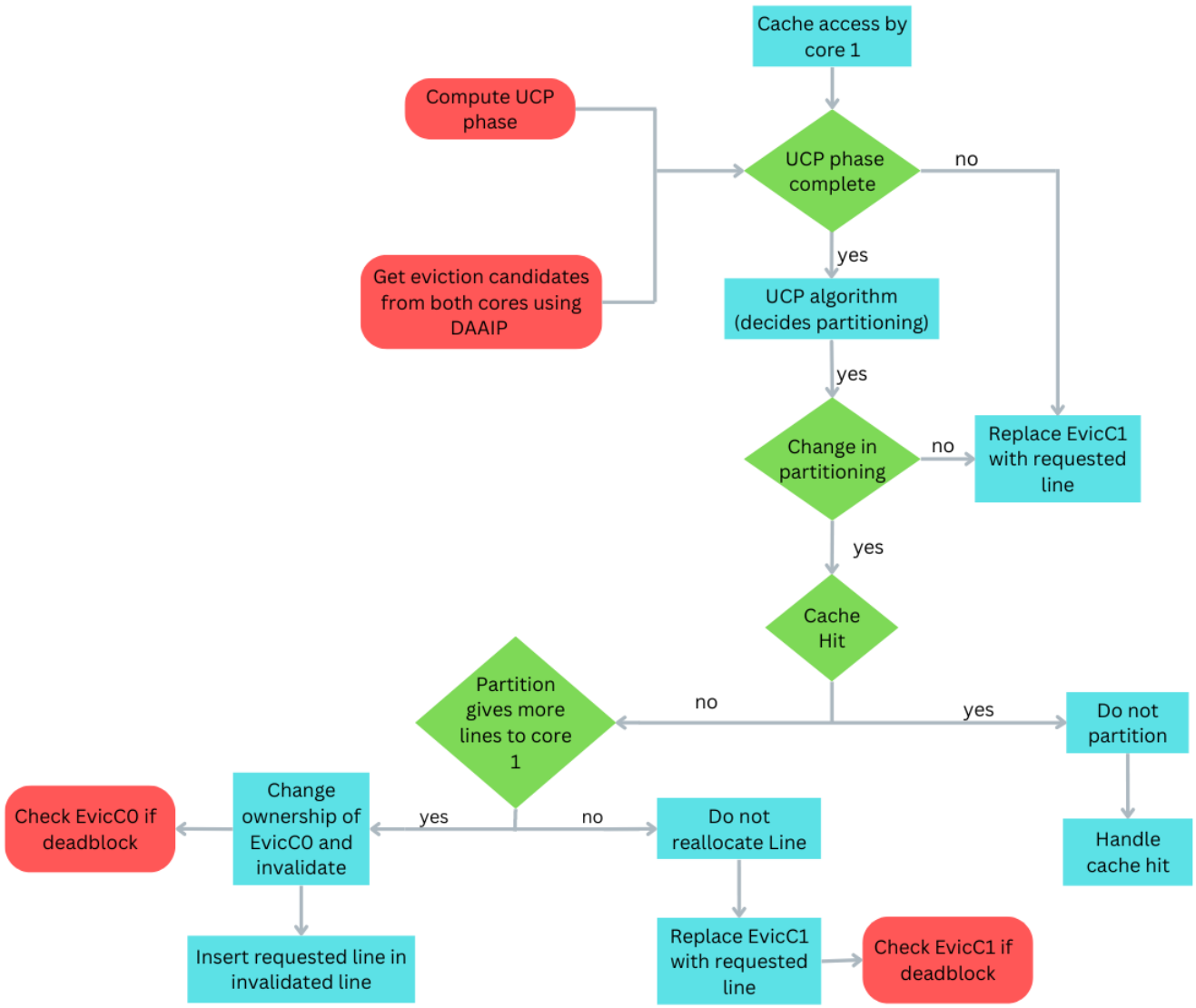

Figure 3.1: Control Flow graph demonstrating proposed idea

Chapter 4

Simulation Results

The simulation was run against 25 pairs of SPEC benchmark programs. Of these, 7 were memory-compute intensive programs, while the rest were memory-memory intensive. The IPC of each core was summed to give the net IPC, and its average over the 25 benchmarks was used for comparisons.

4.0.1 Comparison with PASS-P for memory-memory programs

The configuration for the comparison is as follows

Last Level Cache (LLC)	L3
Number of Cores	2
Threshold fraction (f) (PASS-P)	0.75
Cache Size	4MB
Associativity	4, 16
Threshold	1k

Table 4.1: Configuration details

The results of the two algorithms for memory-memory intensive benchmarks (18) were plotted, and was observed that PASS-P had a minor performance benefit

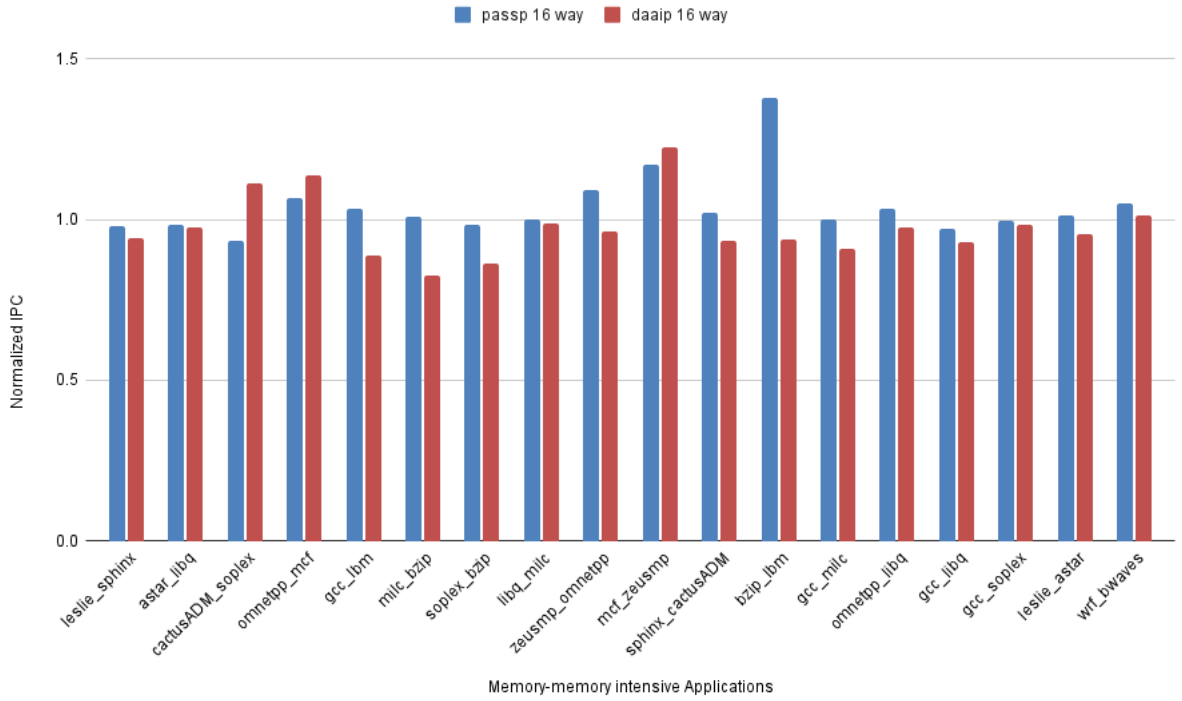

Figure 4.1: performance comparison of PASS-P and modified PASS-P for memory-memory intensive programs (16 way)

4.0.2 Comparison with PASS-P for memory-compute programs

The results of the two algorithms for memory-memory intensive benchmarks (7) were plotted and was observed that PASS-P had an even smaller performance benefit accounting due to the fact the compute-intensive programs do not rely too much on faster memory access.

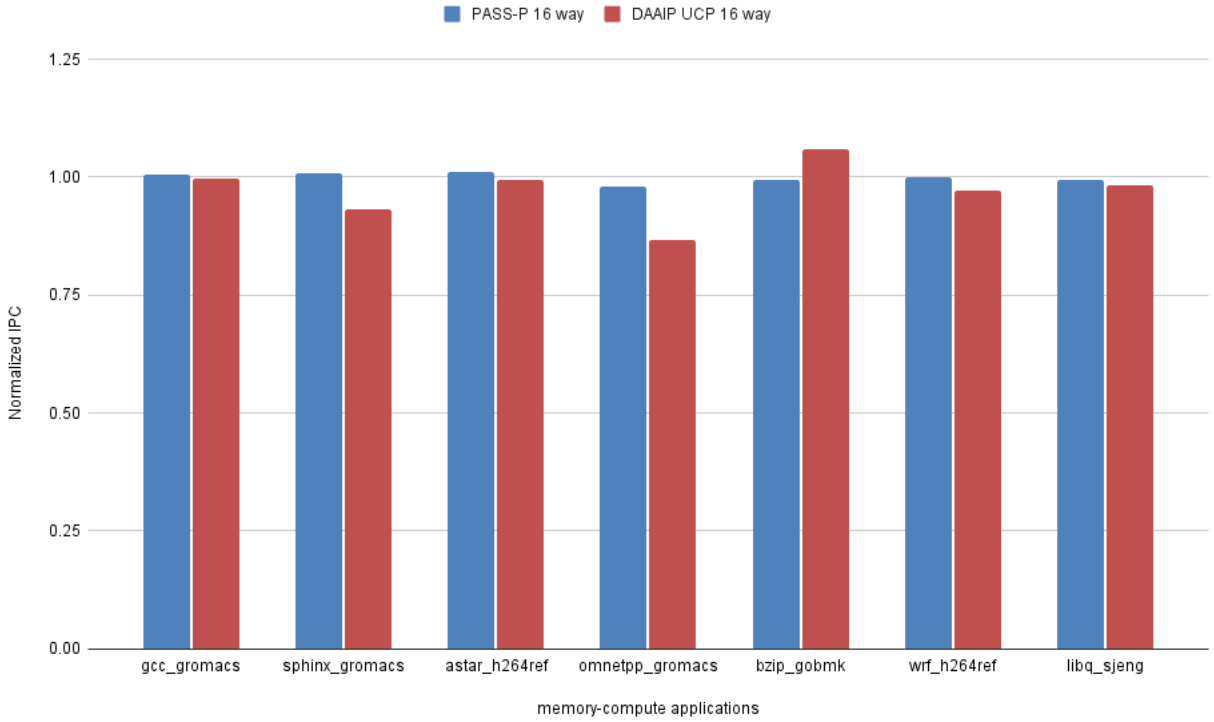

Figure 4.2: performance comparison of PASS-P and modified PASS-P for memory-memory intensive programs (16 way)

The average IPC overall 25 different benchmarks were used to compare the performance between the two algorithms. The simulation results between PASS-P and UCP with DAAIP and invalidation is shown below

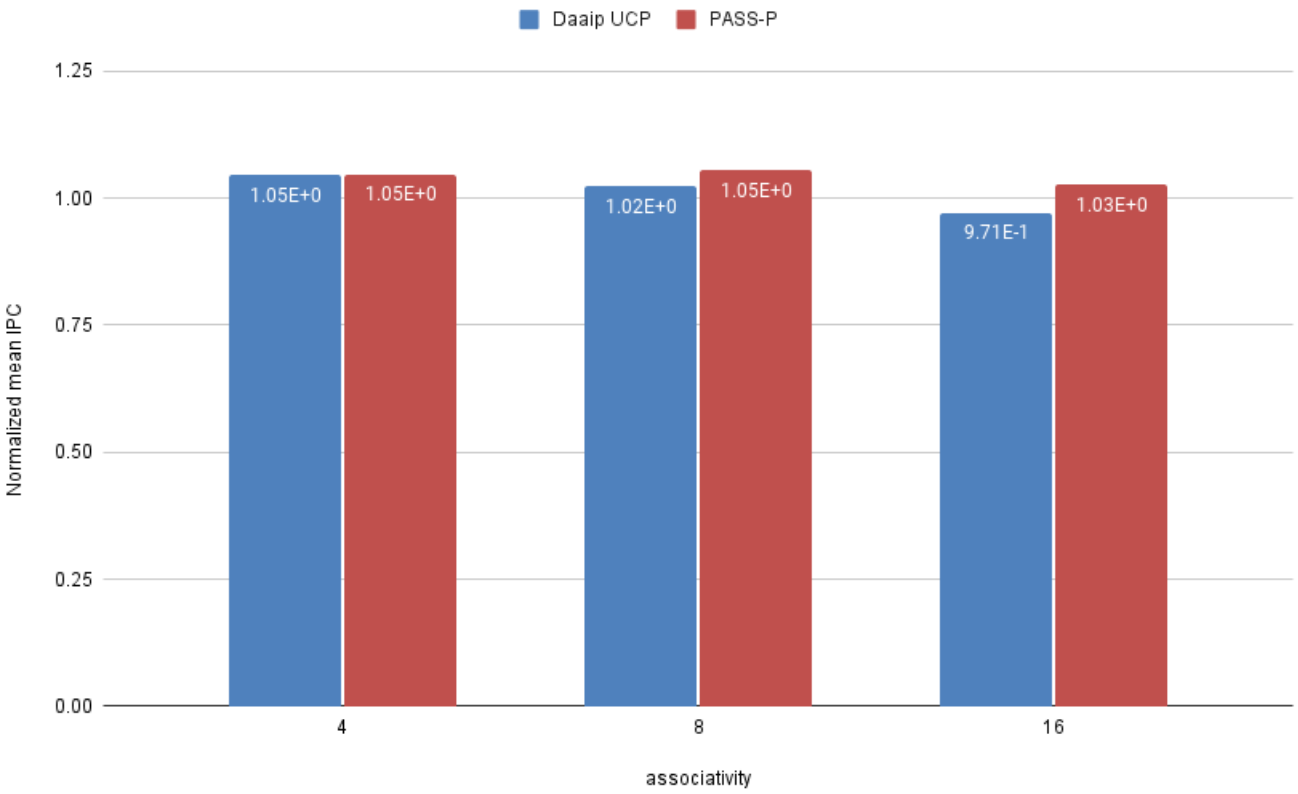

Figure 4.3: performance comparison of PASS-P and modified PASS-P

PASS-P performed better than the modified version in all associativities sparing 4 way. This could be due to PASS-P reallocating only clean lines during partition. This avoids writebacks to main memory post invalidation. Thus writeback latencies are avoided and line requests are catered to faster.

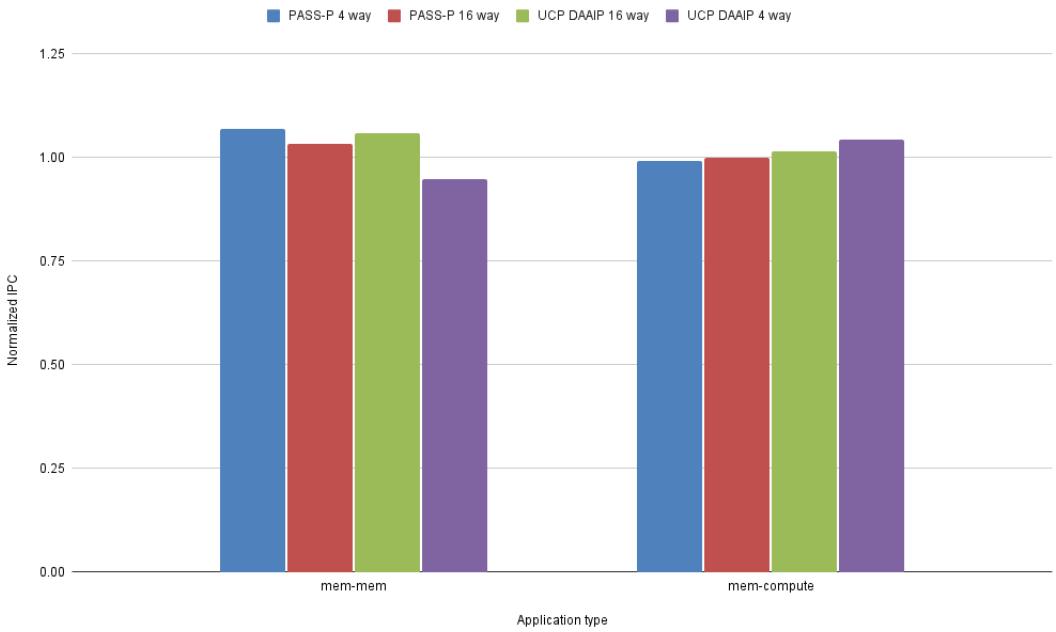

Figure 4.4: performance comparison of PASS-P and modified PASS-P

4.0.3 Partition analysis

The configuration of the system while performing this experiment is tabulated below

Last Level Cache (LLC)	L3
Number of Cores	2
Cache Size	4MB
Associativity	4, 8, 16
Threshold	1k

Table 4.2: Configuration details

The comparison between an entire cache partitioning and a set-wise partition were also obtained.

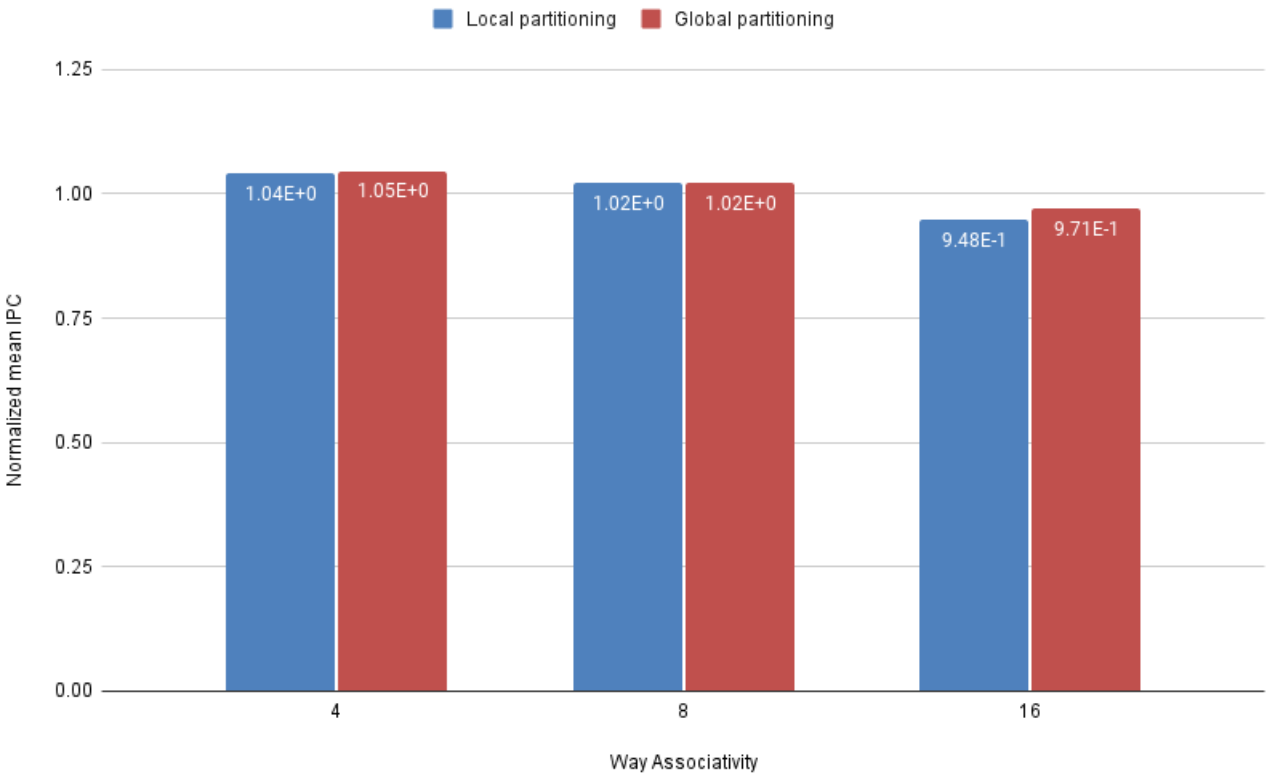

Figure 4.5: Set wise (local) partition vs Uniform (Global) partition

A uniformly partitioned set performed better than the one partitioned set wise. This was due to the partition function being called a fewer number of times in the set wise partition. This results in a partition that does not adapt to the utility of the cores effectively. This is because the partition function is only called for a set once every threshold number of set accesses, and as this is a relatively low number, the partition is done less frequently.

4.0.4 Threshold analysis

The configuration of the system while performing this experiment is tabulated below

Last Level Cache (LLC)	L3
Number of Cores	2
Cache Size	4MB
Associativity	16
Threshold	1k, 10k, 100k, 1M

Table 4.3: Configuration details

The IPC was also compared for different threshold values in UCP with invalidation and DAAIP[4]. The performance was better when UCP partition was once every 10k set accesses.

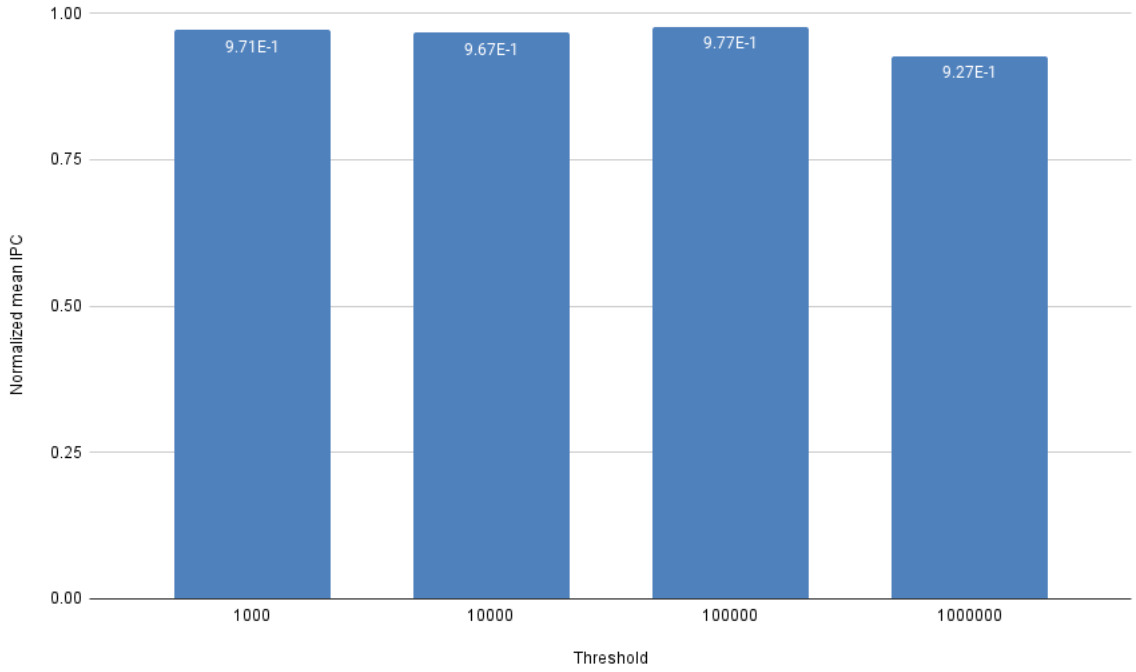

Figure 4.6: performance comparison of PASS-P and modified PASS-P

Chapter 5

Conclusion

The proposed idea was successfully implemented on the SNIPER simulator. The analysis of the same was done thoroughly and was compared against PASS-P. The algorithm was tested for different parameters keeping others constant, and the results were observed. There was extensive examination performed to ensure UCP and DAAIP[4] protocols were followed. The algorithm was studied for a modified version of UCP and a threshold analysis was also done. The results showed that for a low hardware budget, a performance within 5% of PASS-P was obtained.

5.1 Completed Tasks

1. Covered literature on cache hierarchy, shared cache and multi-core processors
2. Extensively analyzed existing literature on side channel attacks and their mitigation
3. Extensively analyzed existing literature on Last Level Cache (LLC) optimizations
4. Covered literature on more efficient replacement policies such as RRIP[5], Hawkeye[6], Mockingjay[7], and DAAIP[4]
5. Read SNIPER implementation of LRU, SRRIP[5], DRRIP[5], MRU, PLRU, NMRU, DAAIP[4] and NRU
6. Implemented the proposed idea as well as PASS-P on SNIPER simulator and obtained results.
7. Performed a comparative study on the results of the two algorithms.

5.2 End Goals

1. Perform a comparative study if the proposed idea on different variations of the cache configurations.
2. Theorize the cause for the obtained results between PASS-P and the proposed idea.

5.3 Future Plans

1. Further explore harware attacks and develop mitigations for the same.
2. Experiment the proposed idea with more effective repacement policies
3. Experiment the same algorithm that reallocates only clean lines when partitioning

References

[1] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, pp. 605–622, 2015.

[2] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (USA), p. 719–732, USENIX Association, 2014.

[3] N. Boran, P. Joshi, and V. Singh, "Pass-p: Performance and security sensitive dynamic cache partitioning," in *Proceedings of the 19th International Conference on Security and Cryptography - Volume 1: SECRYPT,*, pp. 443–450, INSTICC, SciTePress, 2022.

[4] Newton, S. K. Mahto, S. Pai, and V. Singh, "Daaip: Deadblock aware adaptive insertion policy for high performance caching," in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 345–352, 2017.

[5] G. Jia, X. Li, C. Wang, X. Zhou, and Z. Zhu, "Cache promotion policy using re-reference interval prediction," in *2012 IEEE International Conference on Cluster Computing*, pp. 534–537, 2012.

[6] A. J. Calvin Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2022.

[7] A. J. Calvin Lin, Ishan Shah, "Effective mimicry of belady's min policy," *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.