Modeling and analysis of cache partition technique – PASS-P

R&D Report

Submitted in fulfillment of the requirements

for the completion of EE691

by

Anubhav Bhatla

(Roll No. 200070008)

Under the guidance of

Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
2022

Acknowledgement

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

Anubhav Bhatla
Electrical Engineering
IIT Bombay

Abstract

Multicore processors are widely used in today's computing systems, including cloud servers and mobile devices. For such processors, the last-level cache is often shared by the multiple cores. Each core can run its own applications simultaneously, putting pressure on the memory system to efficiently meet their diverse data demands. This leads to data conflicts and one core may end up evicting another core's cache lines. Such negative interference leads to costly off-chip memory accesses (100-400 cycles) and reduces system performance.

Recent work in the field of hardware security has shown security vulnerabilities in such system, particularly various timing channel attacks based on cache interference. These attacks impose a serious threat on modern computing systems and need to be defended against.

Side-channel attacks use differential cache access timing-analysis on lines modified by the victim process. The attacker is able to deduce the addresses accessed by the victim due to the difference in memory access latency in cases of cache hit and cache miss.

PASS-P [1] proposes a Modified-LRU reallocation policy which invalidates reallocated blocks to maintain security. This report aims at performing an extensive analysis of the same and observe sensitivity to different cache parameters.

Contents

List of Figures

Chapter 1

Introduction

In the early 2000s, single-core processors started to see very little improvement in performance with each new generation. This could be explained by the fact that processor frequencies were not able to see a considerable increase due to power considerations. This has led to processors moving to multi-core systems where multiple applications could run simultaneously on each of these cores.

1.1 Cache in Multi-core systems

Cache memory is a high-speed memory type that acts as a buffer between the main memory and the CPU. It reduces the average time to access data from the main memory. The cache is a smaller and faster memory that stores copies of the data from frequently used main memory locations. In the latest computing devices, caches are divided into 3 levels: level-1 cache (or L1), level-2 cache (or L2) and the level-3 cache (or L3). With each increasing level the cache size increases along with the access time.

The L1 cache is optimized to be fast, but it's not typically very large. In a multi-core system, each core usually gets its own L1 cache. The next level, L2 cache, is a bit bigger and usually slower than the L1 cache. L2 cache may be shared between cores or dedicated to each core. The last level cache (LLC), or L3 is most typically shared between cores. This means that the data fetched for one core is also available to other cores.

1.2 Side-Channel Attacks

This sharing of data between the cores leads to the threat of side-channel attacks. Side-channel attacks are a classification of hardware-security attacks that focus on stealing information indirectly by exploiting unintended information leakages. As the name suggest, side-channel attacks do not obtain information by stealing it directly. Instead, they use various "side-channels" such as:

- *Power*: All electronic devices are supplied power through a power rail. In a power-based side channel attack, an attacker would monitor a device's power rails during operation for either current draw or fluctuations in voltage to steal information.

- *EM Radiation*: As Faraday's Law states, electric currents generate a corresponding magnetic field. An EM-based side channel attack leverages this fact by monitoring the EM radiation emitted from a device during operation to steal information.

- *Timing Attacks*: In cryptographic implementations, different mathematical operations may take varying amounts of time to compute based on inputs, key values, and the operation themselves. Timing attacks seek to exploit these timing variations to steal information.

These attacks impose a serious threat on modern computing systems and need to be defended against. In order to mitigate such cache-based side-channel attacks, two types of approaches are used: cache randomization and cache partitioning. Since PASS-P focuses on the second approach, I will confine my analysis to cache partitioning approaches for channel security.

1.3 Cache Partitioning

Previous work [2] proposes *static partitioning* to protect against cache-based side-channel attacks. Within each cache set, it enforces a fixed partitioning of the lines. Each application can only use its own partition of the cache, thus effectively eliminating any cache interference. However, static partitioning comes with a heavy performance penalty because many lines in the cache set remain under-utilized. Cache access behavior of a program can change during run-time, and static partitioning fails to adapt to this change.

To improve performance in a multi-processor system, several *dynamic cache partitioning* (DCP) methods have been proposed. Unfortunately, previous dynamic partitioning techniques are still vulnerable to cache timing channel attacks because the partitioning policy relies on the run-time behavior of each application. Consequently, a malicious application can deduce the addresses accessed by the victim by observing the difference in memory access latency in cases of cache hit and cache miss.

Utility-based Cache Partitioning (UCP) [3] dynamically partitions the LLC in order to maximize the total utility of cache lines for all the running processes. As shown in Figure 1.3, the utility monitors send the information collected is used by a partitioning algorithm to decide the amount of cache resources allocated to each application.

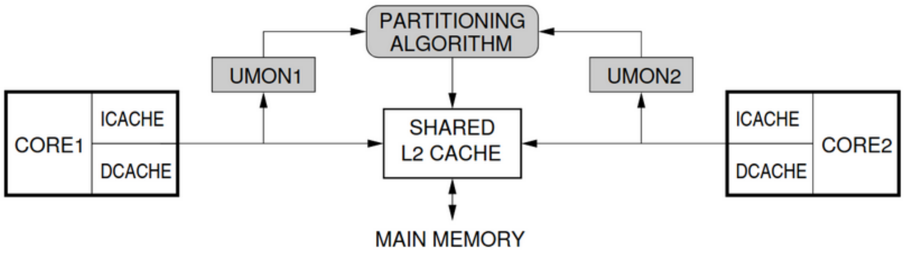


Figure 1.1: Framework for Utility-based Cache Partitioning

Chapter 2

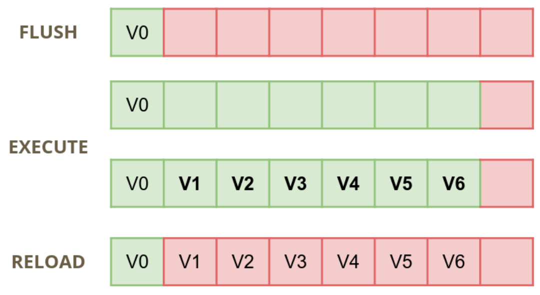
Side-Channel Threat

Dynamic cache partitioning schemes can still be vulnerable to side-channel attacks such as Flush+Reload [4] and Prime+Probe [5] since an attacker application can influence the cache partitioning decisions. In UCP, for example, an attacker program can artificially increase or decrease its utility to cause reallocation of cache lines to and from itself respectively. The mechanism for mounting Flush+Reload [4] attack is given below:

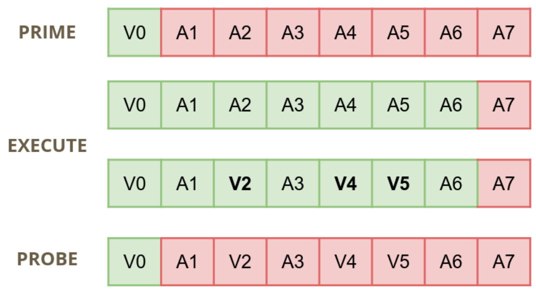
1. *Flush*: The attacker takes all but one cache lines of every set by increasing its utility and flushes them as shown in in Figure 2(a).

2. *Execute*: The attacker returns all the flushed lines to the victim by decreasing its utility. It then waits for the victim to execute as shown.

3. *Reload*: Attacker takes all but one lines by increasing its utility again and reloads addresses of interest. A cache hit or miss on these addresses is indicative of the victim's memory accesses.



(a) Flush+Reload



(b) Prime+Probe

Figure 2.1: Mechanism for mounting side-channel attacks under traditional UCP

Similarly, the mechanism for mounting Prime+Probe [5] attack is given above:

1. *Prime*: The attacker takes all but one cache lines of every set by increasing its utility and primes them with its own data as shown in Figure 2(b).

2. *Execute*: The attacker returns all the primed lines to the victim by decreasing its utility. It then waits for the victim to execute as shown.

3. *Probe*: Attacker takes all but one lines by increasing its utility and reloads the addresses that were previously primed. A cache hit or miss on these addresses is indicative of the victim's memory accesses.

Chapter 3

Literature Survey

LRU replacement policy is commonly used for management of shared last-level cache but recent studies have showed it to be sharing-obliviious. LRU policy is suitable for applications which show a high-degree of data locality i.e. applications which are cache-friendly. However, streaming applications which have very little data reuse perform poorly with LRU. LRU policy makes inefficient use of shared caches for application mixes which are a combination of cache-friendly and streaming applications as the policy treats each cache line independently and doesn't learn from application's past cache reuse behaviour.

Static Re-reference Interval Prediction (SRRIP) [6] helps improve performance of loads that exhibit a distant re-reference interval perform. Results show that SRRIP outperforms LRU by an average of 7% and naturally extends to shared caches and can minimize cache contention between applications with varying memory demands.

Utility-based Cache Partitioning (UCP) [3] dynamically partitions the LLC in order to maximize the total utility of cache lines for all the running processes. This leads to average performance gain of 11% over LRU for a dual-core system.

Deadblock Aware Adaptive Insertion Policy (DAAIP) [7] is also capable of dynamically adapting to the changing cache behaviour of applications sharing the LLC. It takes feedback from the evicted cache blocks to decide if the evicted block was re-used or not between its insertion and eviction. Using this re-usage feedback from each evicted block, DAAIP makes different predictions about re-reference interval for incoming blocks of the different workloads. This results in an average performance improvement of 5.8% over LLC caches managed by SRRIP replacement policy.

Unfortunately, the above dynamic partitioning techniques are vulnerable to cache timing channel attacks. *Static partitioning* [2] is the ideal model to protect against cache-based side-channel attacks as it enforces a fixed partitioning of the lines, forcing applications to use their respective partitions of the cache. This eliminates any cache interference. However, static partitioning comes with a heavy performance penalty due to under-utilization of lines. It also fails to adapt to the dynamic nature of programs.

Performance and Security Sensitive Dynamic Cache Partitioning (PASS-P) [1] proposed a Modified-LRU reallocation policy with cache-line invalidation to overcome the security vulnerability in DCP protocols like Utility-based Cache Partitioning. Due to this, PASS-P overcomes the security vulnerability in DCP protocols such as UCP while gaining an speedup of 7.12% compared to static partitioning.

Chapter 4

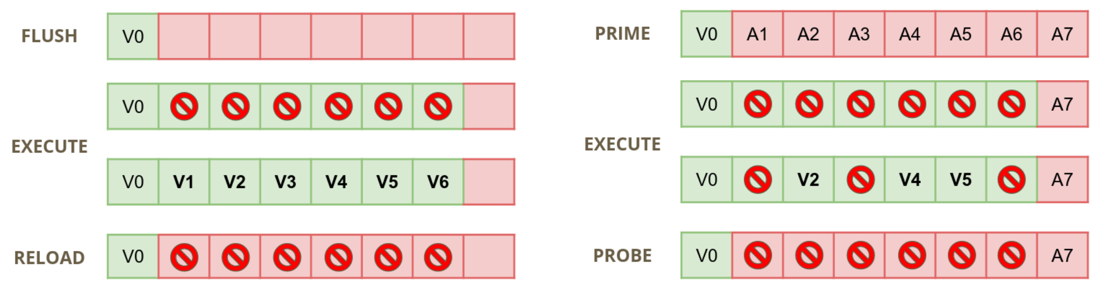
Analysis of PASS-P

PASS-P proposes invalidation of all cache lines that are reallocated from one core to another. This prevents side channel attacks to be mounted in such systems due to the following reasons:

1. *Flush+Reload*: All lines reallocated to the attacker after the Execute step are invalidated, as shown in Figure 4(a) and the attacker gets a miss for every targeted address in Reload step.

2. *Prime+Probe*: The lines primed by the attacker in the Prime step are invalidated when they are reallocated to the victim, as shown in Figure 4(b). Hence, in the Probe step the attacker will get a cache miss for all these invalidated lines.

The attacker's differential timing analysis fails because all addresses that the attacker attempts to fetch result in a cache miss.



(a) Flush+Reload (b) Prime+Probe

Figure 4.1: Invalidating reallocating blocks ensures security against side-channel attacks

The invalidation of cache lines leads to increased miss rate and thus performance drops. In order to counter this, PASS-P proposed a Modified LRU reallocation policy as follows: *Reallocate LRU-Clean line from a set if one exists and only if the clean line is in the f fraction of the least recently used lines allocated to the process, else reallocate the (dirty) LRU line.*

Algorithm 1: Choosing a line for reallocation

```
Input: $List\ L < blockIndex,\ dirtybit >,\ f,\ associativity\ n$
Output: $blockIndex$
Function getReplacementIndex(Input):
    l = getLRUCleanPosition(L);
    if ($l! = null \wedge l \leq f * n$) then
        return l ;                                  // LRU Clean line
    else
        return 0 ;                                  // LRU line
```

4.1 Implementation

In order to implement the reallocation algorithm proposed by PASS-P, I have used the following procedure:

- On a cache access by one of the cores, if we get a hit, the access counter for that particular index is incremented and the block is moved to the MRU position by setting the RRPV to 0.
- In case we get a cache miss, along with incrementing the access counter, we also check the access counter against the threshold value. If the threshold is crossed, we call the UCP partitioning function which recalculates the number of ways to be allotted globally to each core. The access counter is reset to 0.
- We now need to find an eviction candidate. In case the set contains an invalid block, it is returned as the eviction candidate and block is brought to MRU position. We also assign the owner of the block to be the accessing core.
- In case we don't have any invalid blocks in the set, we can end up with the following possibilities:
    1. In case the ways for core-0 decided by UCP is more than the current number of blocks held by the core in the set and the incoming block also belongs to core-0, we reallocate a block from core-1 to core-0. The block to be reallocated is decided using Algorithm 1. This eviction candidate from core-1 is now returned, it's RRPV value set to 0 and the new block brought to MRU position.
    2. Similarly, if the ways for core-1 decided by UCP is more than the current number of blocks held by the core in the set and the incoming block also belongs to core-1, we reallocate a block from core-0 to core-1. The block to be reallocated is decided using Algorithm 1. This eviction candidate from core-0 is now returned, it's RRPV value set to 0 and the new block brought to MRU position.
    3. In any other case, the accessing core decided the eviction candidate. In case core-0 is trying to insert a block, we return an eviction candidate from core-0 using Algorithm 1. Otherwise, we return the eviction candidate from core-1. In either case, the incoming block is inserted at the MRU position.

4.2 Simulation

My simulations include 21 different benchmarks, selected from the SPEC CPU2006 benchmark suite, which have been divided into 14 pairs of both cores running Memory-intensive benchmarks (M-M pairs) [8] and another 7 pairs with one core running a Memory-intensive and the other core running a Compute-intensive benchmark (M-C pairs) [8]. Further, I have assumed that the 3 parameters: threshold fraction (f), threshold (thr) and the cache associativity/size are independent of each other.

4.2.1 Sensitivity to Threshold fraction (f)

In the first set of simulations, my goal is to find the optimal value for f for best performance. The configuration used is as follows:

Last-level Cache (LLC)	L3
Cache Size & Associativity	4MB, 16-way
Threshold (thr)	10k accesses
Threshold fraction (f)	0, 0.25, 0.5, 0.75, 1

The results for the above mentioned configuration are as follows:

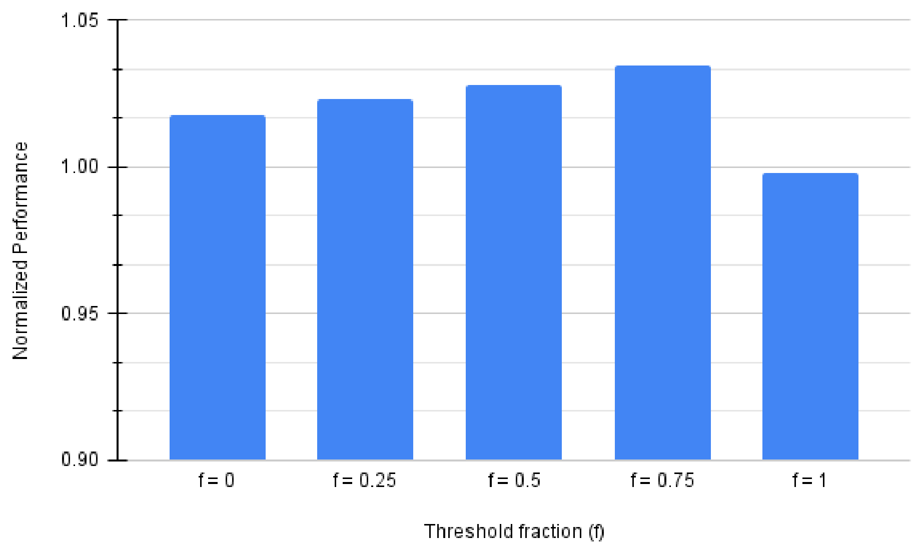


Figure 4.2: Average Overall IPC for various benchmarks for different values of f

Figure 4.2.1 shows that we get the best Average Overall IPC for $f = 0.75$. The case with $f = 0$ corresponds to the regular LRU reallocation policy, resembling the one conventionally used by UCP. We observe that the $f = 0.75$ configuration performs about 2.2% better than the base LRU configuration ($f = 0$). The case when $f = 1$ indicates a policy that always reallocates clean lines even if it is at the MRU position (if clean line exists). Since $f = 0.75$ performs the best, we shall continue with this value for the threshold fraction for the remainder of the simulations.

4.2.2 Sensitivity to Threshold (thr)

The threshold (thr) determines the number of cache accesses after which the UCP partition function is called to re-access the ways to be allocated to each core globally. For these sets of simulations, I shall be using the following configuration:

Last-level Cache (LLC)	L3
Cache Size & Associativity	4MB, 16-way
Threshold (thr)	1k, 10k, 100k, 1M accesses
Threshold fraction (f)	0.75

The results for the above mentioned configuration are as follows:

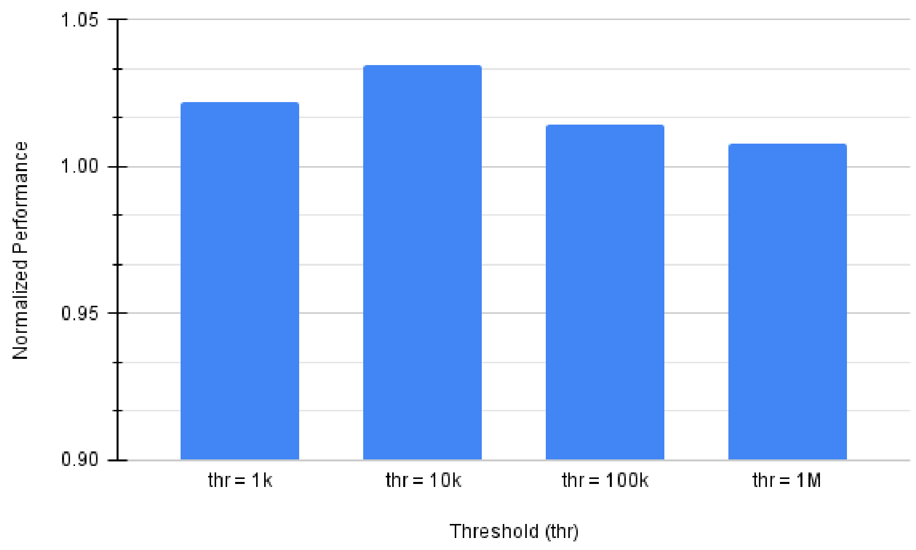


Figure 4.3: Average Overall IPC for various benchmarks for different values of *thr*

Figure 4.2.2 shows similar trends as before. The M-C pairs are not as sensitive to the threshold as compared to the M-M pairs, which give the best Average Overall IPC for $thr = 10k$. Since this configuration performs the best, we shall continue with this value for the threshold for the remainder of the simulations.

4.2.3 Performance Gain Analysis

Now that we have determined the optimal threshold fraction (f) and the threshold (thr) to be 0.75 and 10k accesses respectively, we will now analyze the performance of PASS-P with respect to Static Partitioning using the following LLC configuration:

Last-level Cache (LLC)	L3
Cache Size & Associativity	4MB, 16-way
Threshold (thr)	10k accesses
Threshold fraction (f)	0.75

The results are as follows:

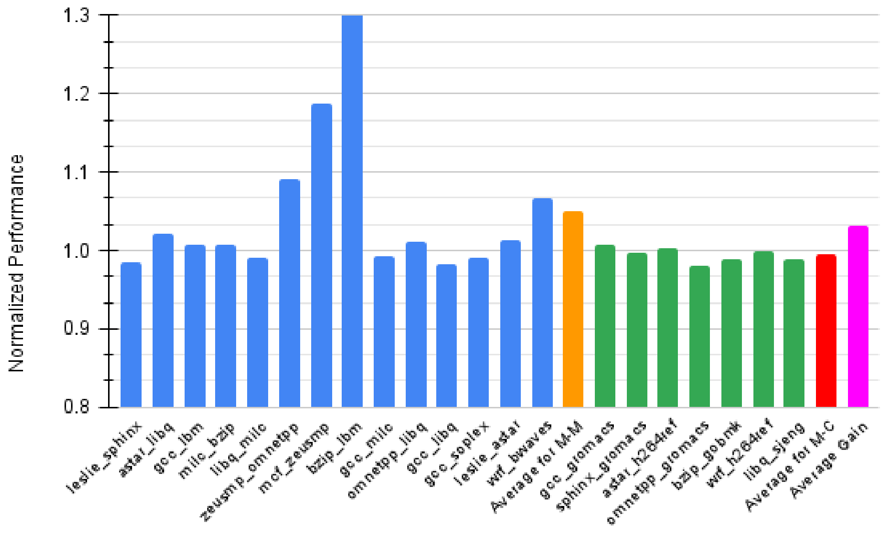


Figure 4.4: Comparison of PASS-P (16-way) normalized to static partitioning for different benchmark pairs (Blue - MM pairs; Green - MC pairs)

Figure 4.2.3 shows a gain of upto 43% and 5.56% on average for M-M benchmark pairs with respect to static partitioning. The overall performance gain for all types of combinations is 3.53%. The almost similar performance to static for the combination of MC pairs is expected, because compute-intensive programs do not have a high utility of the cache. The choice of the cache partitioning protocol does not affect compute-intensive programs' performance as much.

4.2.4 Sensitivity to Cache Associativity

We now try to understand the sensitivity of PASS-P performance gain for different cache associativity, keeping the cache size to be constant (4MB). The results are as follows:

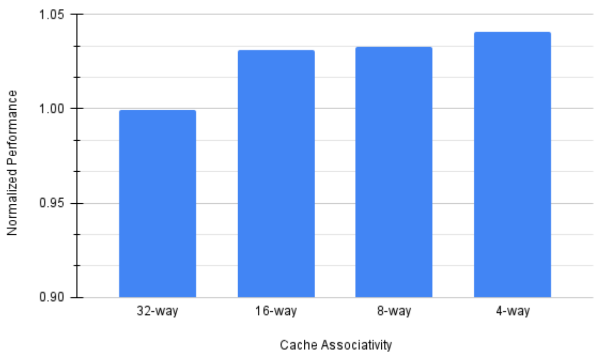


Figure 4.5: Comparison of PASS-P (4MB) normalized to static partitioning for different benchmark pairs and varying associativity

Figure 4.2.4 shows that the performance gain for PASS-P with respect to static partitioning increases with decreasing associativity. This is because lower associativity systems benefit more from PASS-P reallocation policy due to more efficient utilization of the ways.

4.2.5 Sensitivity to Cache Size

We now look at the sensitivity of PASS-P performance gain for different cache sizes, keeping the cache associativity to be constant (16-ways). The results are as follows:

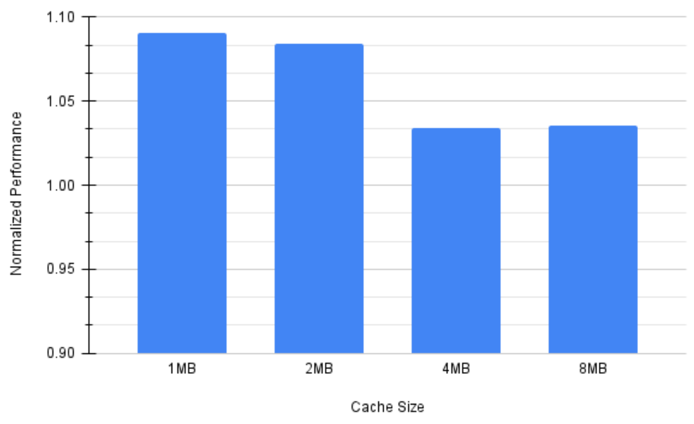
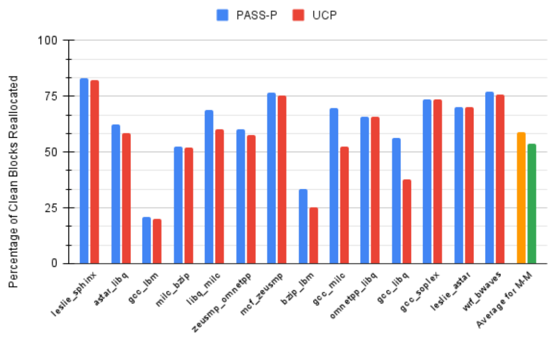


Figure 4.6: Comparison of PASS-P (4MB) normalized to static partitioning for different benchmark pairs and varying cache sizes

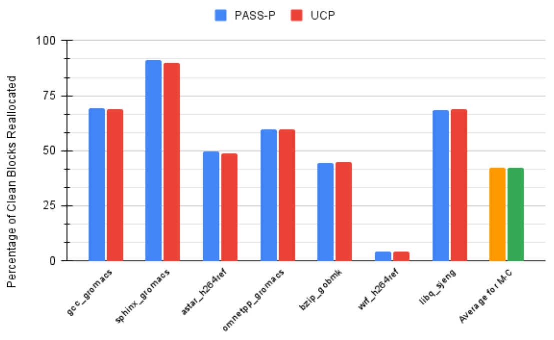
Figure 4.2.5 shows that the performance gain for PASS-P with respect to static partitioning decreases with increasing cache sizes. This is because the applications can now better fit their working sets into the cache, leading to reduced cache conflicts. This leads to a decrease in performance gains as the dependence on the reallocation policy reduces.

4.2.6 Clean Block Analysis

We now try and look at how PASS-P's clean block preference for reallocation affects the percentage of reallocated clean blocks compared to traditional UCP:



(a) M-M pairs



(b) M-C pairs

Figure 4.7: Percentage of clean reallocated blocks for PASS-P and UCP

Figures 4.2.6 shows an increase in clean reallocated blocks from 53.8% to 59.1% as we move from UCP to PASS-P for the M-M pairs. This leads to reduced memory traffic during reallocation and thus helps improve IPC. The M-C pairs, however, don't show a notable difference in clean block percentage due to their insensitivity to the reallocation policy.

4.2.7 Deadblock Analysis

Deadblock is the term used for a block which does not get a single access in the time from being brought into the cache to when it is evicted to make way for an incoming block. We now look at the percentage of deadblocks inserted under PASS-P:

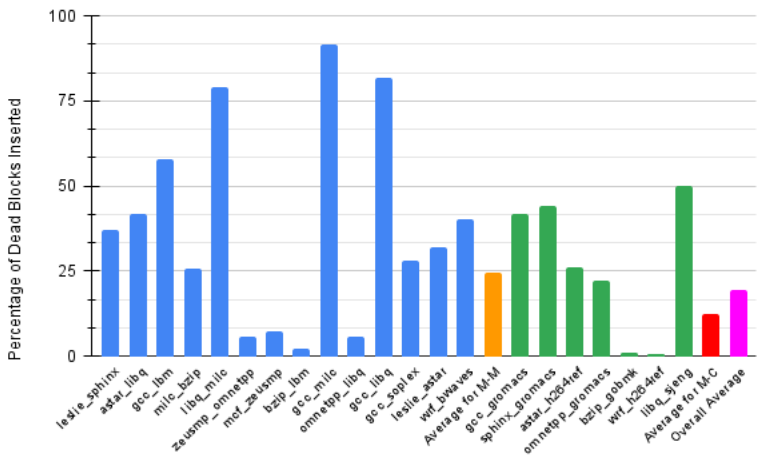


Figure 4.8: Percentage of Deadblocks inserted under PASS-P

Figure 4.2.7 shows that 14 out of the 21 benchmark pairs show a deadblock percentage of more than 25%. This can be exploited for the purpose of reallocation. The ideal eviction candidate is a block that is not going to be accessed again i.e. it is a deadblock. Therefore we can make use of deadblock-aware policies such as DAAIP [7] to make use of this high percentage of deadblocks inserted and improve performance.

Chapter 5

Conclusion

The PASS-P reallocation policy was successfully implemented and simulated on the SNIPER simulator. An extensive analysis was performed on the sensitivity of PASS-P to various parameters such as threshold fraction (f), threshold (thr), cache associativity and cache size. Similar performance gains with respect to static partitioning were obtained as mentioned in the paper [1]. An extensive analysis was performed on the percentage of clean blocks reallocated as compared to traditional UCP and how it helps improve performance. In the end, the percentage of deadblocks inserted under PASS-P was measured and it was argued that a deadblock-aware policy such as DAAIP [7] could help choose a better eviction candidate and thus improve performance.

5.1 Work done so far

- Covered literature on cache hierarchy, shared cache and multi-core processors
- Covered literature on cache replacement policies
- Extensively analyzed existing literature on side channel attacks and their mitigation
- Became comfortable with the SNIPER simulator by reading existing implementations of LRU, NRU, MRU, PLRU, NMRU, SRRIP [6] and DAAIP [7]
- Implemented PASS-P on SNIPER simulator and obtained results for different cache configurations.
- Performed extensive analysis of reallocated blocks and deadblocks for the different benchmarks.

5.2 Future Work

- Explore various deadblock aware policies for finding a better eviction candidate.
- Explore newer hardware attacks and develop mitigations for the same.

References

[1] 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.

[2] D. Page, “Partitioned cache architecture as a side-channel defence mechanism.” Cryptology ePrint Archive, Paper 2005/280, 2005. `https://eprint.iacr.org/2005/280`.

[3] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[4] 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.

[5] 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.

[6] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA ’10, (New York, NY, USA), p. 60–71, Association for Computing Machinery, 2010.

[7] 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 2017, Boston, MA, USA, November 5-8, 2017*, pp. 345–352, IEEE Computer Society, 2017.

[8] A. Navarro-Torres, J. Alastruey-Benedé, P. Ibáñez-Marín, and V. Viñals-Yúfera, “Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the intel xeon Skylake-SP,” *PLoS One*, vol. 14, p. e0220135, Aug. 2019.