Exploiting Static Partitioning in Modern Cache Replacement Policies.

Ashish Kumar
2021AIM1003
dept. of Computer Science Engineering
Indian Institute of Technology Ropar
2021aim1003@iitrpr.ac.in

Shivam singh
2021CSM1006
dept. of Computer Science Engineering
Indian Institute of Technology Ropar
2021csm1006@iitrpr.ac.in

I. Introduction

Cache reduces the access time between processor and memory. When miss occurs in last level cache processor stalls for hundreds of cycle which results in increasing the access time. To reduce the access time, we need to reduce last level cache miss by managing LLC cache friendly. This totally depends upon how efficient the replacement policy used in last level cache. Since the Advancement in last level cache, Many replacement policies have been proposed. Some of them becomes very popular and change the course of replacement. The basic one is Least recently used (LRU) Replacement Policy which benefits the recent cache lines. Real time applications can be Cache-friendly or Cache-averse based on the demand. Cache-friendly application benefits from the cache, while cache-averse application exploits the cache resources like streaming application and thrashing application. To benefit cache from cache-averse applications many replacement policies such as DIP, RRIP has been proposed. Main limitation of these policies is that they perform better for some kind of application.

For the workloads with a working set greater than the available cache size cache performance can be significantly improved if the cache can retain some fraction of the working set in LLC. Based on above idea DIP replacement policy is proposed. DIP separates replacement policy into two parts Victim selection policy and insertion policy. Victim selection policy decides which line will be evicted for storing an incoming line. The insertion policy decides whether to place an incoming block. The most efficient replacement policy is belady's optimal replacement policy which makes current decision based on future scope of that line, but it is impractical. A hawkeye paper proposed a method to mimic Belady's optimal replacement policy. They used the past information to predict whether current block is hit or miss.

When replacement policies are proposed they thinks that world is innocent but this is not a case. In recent years, Many attacks are happening on last level cache to exploit the replacement policies. One of the famous attack is covert channel attack [3]. In this attack, there are two application running on two core one is spy and other is trojan. These application communicate by exploiting the replacement policy and shared some sensitive information to outside world. These attacks are proposed while thinking that the replacement policy is LRU but these attack can also possible in different replacement policies. Some techniques are proposed to prevent these types of attacks. One of them is partitioning, if we partition the cache so that no application can evict the block of other application. Then spy and trojan application can't communicate with each other and no information will leak. Partitioning of cache can be done by two ways, one is static partitioning and other is dynamic partitioning. In static partitioning, partition size is fixed and set at boot time and remains same untill we change and reboot the system. In dynamic partitioning, Partition can change based on demand of cache by the application.

II. Literature Review

Many interesting and fascinated work has been produced to improve the performance of replacement policies. First state of the art work in this field is Dynamic insertion policy(DIP) [4]. Which focuses on reducing the cache misses for thrashing application by retaining some portion of thrashing application in cache. It dynamically changes the insertion policy by inserting the incoming cache lines at Least Recently Used(LRU) position or Most Recently Used(MRU) Position. It uses DSS mechanism to select insertion policy. When workload have mixed access patterns. They cannot distinguish between distant re-reference interval than near-immediate re-reference interval. RRIP [2] paper suggests to use 2-bit value per cache block called RRPV value. RRPV values set based upon the re-reference interval of a block. Eviction decision based upon the RRPV value. Highest RRPV value block will get evicted first. It predicts that any cache block that receives a hit will have a near immediate re-reference and sets its RRPV value to zero. This RRPV technique benefits the scanning application.

Hawkeye [1] makes a replacement policy as a binary classification problem, where the goal is to determine if an incoming block is cache-friendly or cache-averse. Hawkeye paper mimics the optimal replacement policy and works great for all type of application. It performs better than almost every proposed policies. Hawkeye uses past cache access to predict the current block is hit or miss. It performs better when we retain 8X past cache access. OPT's decision for past cache access is determined at its next use time. To simulate OPT decision, Hawkeye use set duelling to capture long history with very less hardware overhead.

UCP [5] paper proposed a dynamic partitioning scheme based on the demand of application. Each core have a UMON which monitor the demand of cache on a application and dynamically change the partition of LLC. It gives more ways to the cache-friendly application and limit the ways for cache-averse application. UCP helps in improving performance of the system when an cache-friendly application runing with cache-averse application.

III. Proposed work

Our proposed work will exploits the hawkeye replacement policy and try to reduce the performance of an application running on them. We partitioned the cache and run it for two application let's say A and B. Assume B is an attacker application and it flood the request. As it is partitioned cache, Application can not impact directly on application A but can impact indirectly. Hawkeye uses sampler which stored the past access history of incoming lines and partitioning don't apply on sampler. When accessing a cache block of sampling set, If it predicts hit then count value will increase in predictor and if it predicted miss, it decrease the count in predictor. When non-sampling set request comes it looks into predictor corresponding to program counter and assign RRPV value to that block based on count value. Attacker application will target the sampling set which result in eviction of the block of application A from the sampler. So, When the new request come in non-sampling set it look into predictor and assign bad RRPV value to that block. Therefore, attacker floods the core with request such that it remove all the blocks of application A from the sampler. Now, if we run application A with application C, As no past history of application A is present in sampler all incoming block in non-sampling set receives bad/high RRPV value even that block is cache-friendly. Hence, Application A will not benefit from the hawkeye replacement policy and it's performance will degrade.

For starting result, We apply static partitioning on cache and use a single optgen vector for both the partition and run it for 2 cores. After it, We replace the core 2 application with other application and observe the IPC for core 1 application for both run. If IPC of application A degrades means that another application evict the block of application A from sampler and assing bad RRPV value to incomming blocks. Application A unable to benefit from the replacement policy. We run the same traces again but this time we also divides the optgen vector each for their core and after running observe their IPC.

IV. Experimental Setup

We use trace based champsim simulator for the experimental result and cofigure it's parameter as,

- Number of processor = 2
- LLC sets = 4096
- LLC ways = 16
- Block size = 64B
- Warm-up instructions = 50M
- Execution instruction = 100M

We perform result on the traces show on below table and compare their IPC for different set up. Following are the different setup for collecting result.

- Normal Hawkeye.
- Static Partition with 1 Optgen.
- Static Partition with 2 Optgen.
- Dynamic Partition with 2 Optgen.

cactusADM_734B.trace.xz	gamess_247B.trace.xz
cactusADM_734B.trace.xz	mcf_158B.trace.xz
cactusADM_734B.trace.xz	GemsFDTD_109B.trace.xz
gamess_316B.trace.xz	gamess_247B.trace.xz
gamess_316B.trace.xz	mcf_158B.trace.xz
gamess_316B.trace.xz	GemsFDTD_109B.trace.xz
bzip2_281B.trace.xz	gamess_247B.trace.xz
bzip2_281B.trace.xz	mcf_158B.trace.xz
bzip2_281B.trace.xz	GemsFDTD_109B.trace.xz
hmmer_546B.trace.xz	gamess_247B.trace.xz
hmmer_546B.trace.xz	mcf_158B.trace.xz
hmmer_546B.trace.xz	GemsFDTD_109B.trace.xz

V. Results

We run the 4 simulation for all different traces and compare each core IPC value and plot them. Fig1 graph shows the IPC value for different simulation. Series 1 refer to the state when normal hawkeye code runs. Series 2 refer to the state when we static partiton the cache but use only one Optgen. Series 3 refer to the state when we static partition the cache and also divide the Optgen for different partition. In series 4, Dynamic cache partition with two Optgen. We change the partition based on the number of instrucion. The IPC of core 1 application are shown in fig2. Our main focus on core-0 application. We can see a little bit of degradation of IPC of core-0 application. It is little less because there is less cache hits in sampler set. if we make application that target to specific sample sets then this difference can increase and we can see clear difference of IPC. In below table, column-1 shows the hit rate to Optgen vector when normal hawkeye code runs against traces. Column-2 and column-3 shows the hit rate of Optgen-1 and Optgen-2 vector when the cache is statically partitioned. We can clearly see the reduction in hit rate to Optgen vector when we divide the Optgen vector.

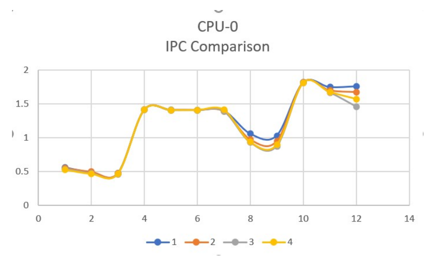

Fig. 1. Types of attack

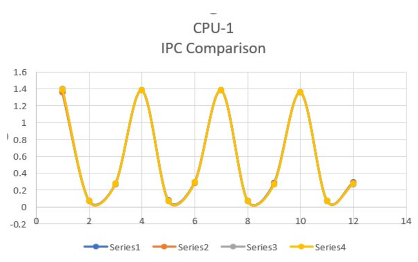

Fig. 2. Types of attack

43.7995	31.5504	0
34.038	5.80115	26.6501
22.2465	5.65957	19.1245
6.36943	0	13.1579
48.4795	16.6667	42.2235
26.4845	3.10559	26.125
84.1221	76.4133	0
55.9235	35.6847	24.4211
46.4201	32.9579	18.2172
89.2288	79.6294	0
56.3239	23.9278	26.799
42.3465	18.3729	18.958

Future Work

We can extend our work by making the application to request the block that map to sampling set. we attack sampling set which result in removing of all the blocks of application A from the sampler that leads to predictor to assign high RRPV value to cache-friendly blocks. We can further modify our work for dynamic cache partition. Here we statically do the dynamic partition we can change it for truly dynamic partition.

Conclusion

We exploits the hawkeye replacement policy. Even though we partition the cache, the application running on one core can indirecly impact other application. We run the simmulation for different setup and find out that attacker application can degrade the performance of victim application by evicting it's history from sampler. The hit rate to a Optgen also decreases when we partition the Optgen. Due to decrease in hit rate to Optgen some blocks of application A will receive bad RRPV value than without partition. Therefore, there is slight change in performance of application running on core-0.

References

[1] Akanksha Jain and Calvin Lin. Hawkeye: Leveraging belady's algorithm for improved cache replacement. *The 2nd Cache Replacement Championship*, 2017.

[2] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH Computer Architecture News*, 38(3):60–71, 2010.

[3] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 481–487. IEEE, 2010.

[4] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, 2007.

[5] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.