EE488:Parallel Computer Architecture 7/6/2020

Professor Myoungsoo Jung

Final Project

A Cache Replacement Policy Based on Re-reference Count

Yonas Girma Kelemework

20170758

I. Introduction

Cache memory is a faster memory introduced between processor and main memory in order to store copies of data from frequently used main memory locations. A cache hit occurs if the required block is found within the cache, else a cache miss is said to occur. The missed block must be brought from the main memory in to the cache in order to service the miss. The technique that the memory management chooses to make room for an incoming block is known as replacement policy. The most efficient replacement algorithm is to always discard the information that will not be needed for the longest time in the future. This policy known as Belady's algorithm or optimal replacement policy is not practical since the references are not known in advance. Thus other algorithms such as LRU(Least Recently Used), MRU(Most Recently Used), LFU(Least Frequently Used) etc.. , that use the past to predict the future are used instead.

Cache performance can be measured in terms of Average Memory Access Time, which is given by,

Average Memory Access Time= Hit Time + Miss Rate * Miss Penalty;

Where, Hit time is the time taken by the CPU to get the contents of its request when it is a hit in the cache. Miss Rate gives the fraction of references that are missed in the cache out of the total references and Miss Penalty is the additional time required to service such a cache miss. L2 cache miss causes the processor to stall for hundreds of clock cycles since the next level of access is main memory. Need for access to main memory leads to memory hierarchy performance reduction. So reduction in L2 cache miss rate is crucial in leading to the system performance improvement. Cache replacement policy plays a significant role in cache miss rate reduction by evicting and making room for appropriate memory blocks. So the major aim of this work is to develop a novel cache replacement policy which helps in reducing L2 cache miss rate.

II. Background

Main memory access causes the processor to stall for several clock cycles. So a faster memory which stores recently used data would improve the overall performance of memory hierarchy. Caches were introduced to serve this purpose. The size of caches is kept small in order to facilitate faster cache access. But smaller the cache size, lesser the data it can contain. Hence, the cache capacity needed to be increased somehow without sacrificing the performance. For this reason, caches are organized in hierarchies, named by their level, usually L1, L2 and sometimes L3. L1 is the first level cache, the fastest and closest to the CPU. If a piece of data is not found in the L1, the request goes to the L2 cache which is larger.

The request to access a main memory block has to be mapped first on to the cache. This mapping between main memory blocks and cache blocks is an important design issue. There are three general approaches for the mapping of a main memory block to the cache. They are direct mapped cache, fully associative cache and set associative cache. In direct mapped approach, a main memory block is mapped strictly on to a cache block. In fully associative approach, a main memory block can be mapped on to any cache block. In set associative approach, a main memory block can be mapped on to any cache blocks of a particular set.

The memory access pattern plays a crucial role in reducing cache miss rate. Several applications exhibit different memory access pattern. The applications can be divided into recency friendly applications, cache thrashing applications, streaming applications and mixed access pattern applications. Recency friendly applications have a near-immediate re-reference interval and benefit from cache and continue to do so even when the cache size is increased. For such kind of applications, LRU is the most suitable replacement policy. Cache thrashing applications have a working set that is greater than the size of the available shared cache and thus cause thrashing under LRU policy. Such kind of applications, work effectively under LIP policy. Streaming applications have extremely large working set and poor cache reuse. This kind of applications cause thrashing, under any policy. Mixed access pattern kind of applications are workloads where some references have a near-immediate re-reference interval while other references have a distant re-reference interval.

Compiler optimization techniques were introduced to reduce the cache miss rate. Some of which were loop interchange, loop fusion, array merging, etc… Programs have nested loops that access data in memory in non-sequential order. Loop interchange simply exchanges the nesting of the loops which make code access the data in the order in which they are stored. Loop interchange reduces misses by improving spatial locality. Loop fusion combines two independent loops that have same looping and common variables. Fusing multiple loops into a single loop allows the data in cache to be used repeatedly before being swapped out. Loop fusion reduces misses through improved temporal locality. Array merging technique improve spatial locality by using single array of compound elements instead of using two different arrays.

Least Recently Used (LRU) policy perform well for workloads having cyclic access pattern which completely fit in cache but cause thrashing for memory intensive workload which does not fit in cache. LIP (LRU Insertion Policy) prevents thrashing for memory intensive workloads by retaining some blocks which hits in cache. LIP uses LRU insertion policy unlike LRU which uses MRU insertion policy. BIP (Bimodal Insertion Policy) uses probabilistic insertion. It adapts to changes in the working set but not optimal for all applications. DIP (Dynamic Insertion Policy) chooses between LRU and BIP depending on which policy incur fewer misses. DIP achieves this by set dueling method. In set dueling some of the cache sets will be dedicated to follow LRU policy and some other sets will follow BIP policy and rest of the sets are known as follower sets. The follower sets choose that policy which incurs fewer misses in the above sets. Re-reference Interval Predictor (RRIP) prevents blocks with a distant re-reference interval from polluting the cache. In this approach, RRIP chain represents the order in which blocks are predicted to be re-referenced. The block at the head of the RRIP chain is predicted to have a near-immediate reference interval while the block at the tail of RRIP chain is predicted to have a distant re-reference interval. On a cache miss, the block at the tail of the RRIP chain will be replaced.

The insertion policy in the LLC can have a significant impact on cache efficiency. However, a fixed insertion policy can allow blocks to remain in the cache longer than necessary, resulting in less efficiency. This work introduces insertion policy selection using Decision Tree Analysis (DTA). This policy uses the fact that the LLC filters temporal locality. Many of the lines brought to the cache are never accessed again. Even if they are re-referenced they do not experience bursts, but rather they are reused when

they are near to the LRU position in the LRU stack. This technique uses decision tree analysis of multi-set dueling to choose the optimal insertion position in the LRU stack. Inserting in this position, evicts those blocks which are never used again while those blocks which might be reused remain in the cache and avoid a miss.

III. Proposed Policy

The overall performance of the memory hierarchy can be improved by optimizing the cache performance. The performance of cache can be optimized by reducing the cache miss rate. Access to last level cache increases latency thereby reducing the cache performance. Hence reducing last level cache miss rate is a major concern.
The efficiency of the cache replacement policy affects both the hit rate and the access latency of a cache system. A replacement policy is decided by the insertion policy, promotion policy and victim selection policy used in it. Once if a cache miss occurs, the insertion policy determines where in the priority level, the incoming cache block has to be inserted. On inserting the new cache block, sometimes an existing cache block has to be removed from the cache. Victim selection policy decides which block needs to be replaced from the cache. On a cache hit, the priority of the block need to be changed which is decided by the promotion policy.

LRU Insertion Policy (LIP) is a modification of LRU in which the insertion policy is modified. In LIP policy the incoming blocks are inserted in the LRU position by retaining those blocks which fit in cache. This work based on re-reference count (RRC) modifies the promotion policy in LIP. In this technique, the promotion policy similar in case of LIP is modified in such a way that if the re-reference count is greater than a threshold value, the cache block is promoted on to the MRU position else it is promoted to LRU position

In case of cyclic workloads, the most recently used block is referenced far later in the future. In LIP, consecutive misses occur due to the replacement of cache block that will be used for the very next access. This policy overcomes this limitation in LIP. Under RRC policy, most recently used block will be replaced on a miss since this block will be the one that will have greater re-reference distance. Thus reduction in cache miss rate is obtained leading to the overall performance improvement. The pseudocode is as given below.

Algorithm 1 Replacement policy using re-reference count
Input: SPEC CPU 2006 Benchmarks
Output: L2 Cache Miss Rate
1: **for** blocks undergone cache hit **do**
2: **if** blocks re-reference count > (14) **then**
3: Move to MRU position
4: **else**
5: Move to LRU position
6: **end if**
7: **end for**
8: **for** blocks need to be inserted to cache **do**
9: Move to LRU position
10: **end for**
11: **for** blocks to be evicted from the cache **do**
12: Select from LRU position

13: **end for**

IV. Simulation and Result

I used gem5 to simulate and evaluate the policy. First since the policy only require a few modification to the LRU policy I copied and pasted the LRU sources code as a skeleton. In the .h file I added a new field for each entry called reference count to record the number of references to each entry. Then in the .c source code there are four main functions invalidate(), touh(), reset(), and getVictim(). In invalidate(), I modified the code and added a statement to invalidate the reference count which resets the value to 0. In touch(), where we need to update replacement values, I added the logic in the pseudocode above from line 2-6, where I update the reference count by adding one, then put in an if statement where if the reference count is above a threshold I will set the tick value for the entry to current tick value making it the MRU entry, else I will set the tick value to initial value making it the LRU entry. In the reset(), which is called when a value is inserted I increase the reference count by one and set the tick value to initial tick which makes it LRU entry. Lastly, the getVicitim code both for the LRU and Re-reference based policy is the same thus I didn't add any modifications and entries are evicted based on their tick value.

After modifying the code I create a class object for the policy, added the .c file to the cache configuration script and compiled it to run different tests.

Simulation Configuration

	L1 Icache	L1 Dcache	L2 cache
Associativity	8	8	16
Size	32Kb	32Kb	1Mb

The benchmarks I used are benchmarks we have been using during the course. For reference these are:

Wl1	test_bench/2MM/2mm_base'
Wl2	'test_bench/BFS/bfs','-f','test_bench/BFS/USA-road-d.NY.gr
Wl3	'test_bench/bzip2/bzip2_base.amd64-m64-gcc42-nn','test_bench/bzip2/input.source','280'
Wl4	'test_bench/mcf/mcf_base.amd64-m64-gcc42-nn','test_bench/mcf/inp.in'

Although it was possible to set the up the configurations such that threshold value to be controlled from our config file I was not able to add that feature and had to change the threshold manually in the .c code and rebuild gem5 to run simulations on various threshold values.

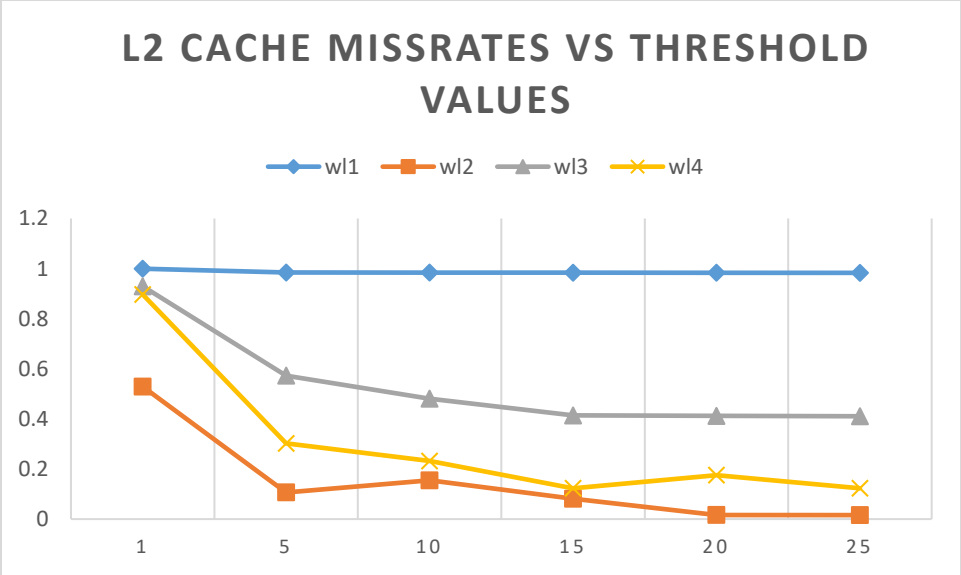

Figure 1: L2 cache miss-rate for various number of threshold values.

As we can see in the figure above as the threshold values is increased the L2 cache miss-rate decreases consistently with some variations depending on the workload, thus from the above result we can conclude that the shorter we are delaying the eviction of a recently referenced value the better lower the miss-rate is going to be thus we can conclude that most of the entries in L2 are not re-referenced immediately, and L2 caches filter temporal locality and stress on spatial locality.

My next I used a threshold value of 10 to be conservative and compared the L1 miss-rates, L2 miss-rate and IPC ratio(re-reference/LRU) between re-reference policy and LRU policy.

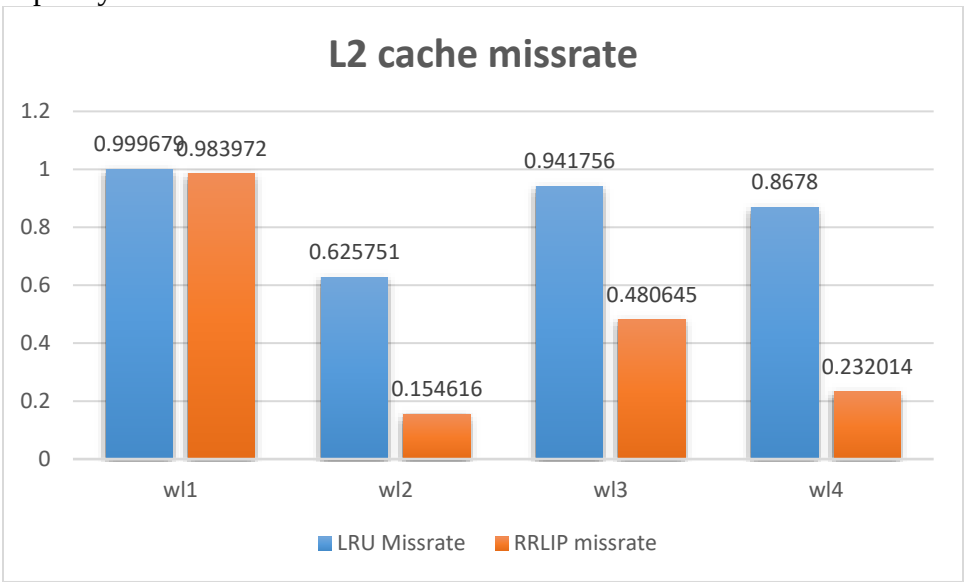

Figure2: L2 miss-rate for LRU and Re-reference missrate.

From the Figure 2 we can see that the L2 miss-rate is reduced significantly for all the workloads. For the first workload the reduction is just 1 percent but for the rest(Wl2, 47%; Wl3, 46%; Wl4, 63%). Hence the new policy has successfully been optimized to

take advantage of spatial locality. The paper I am trying to emulate just provides the L2 miss-rate reduction and doesn't discuss the L1 misstates and IPC results. So I decided to look further into the data and check if the paper delivers what it promises (reduced L2 cache miss-rate=better overall performance).

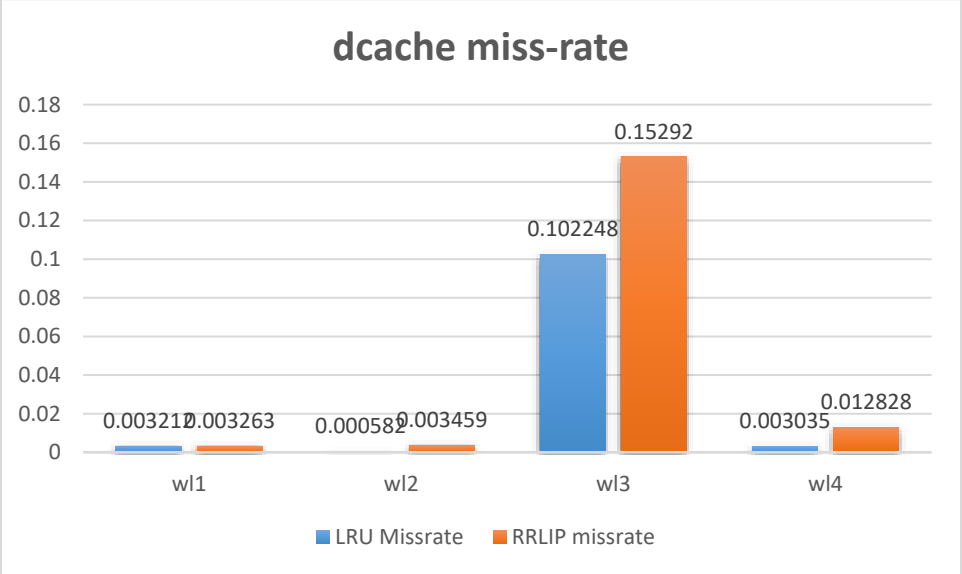

Figure 3: Dcache miss-rate L1-level for LRU vs RRLIP

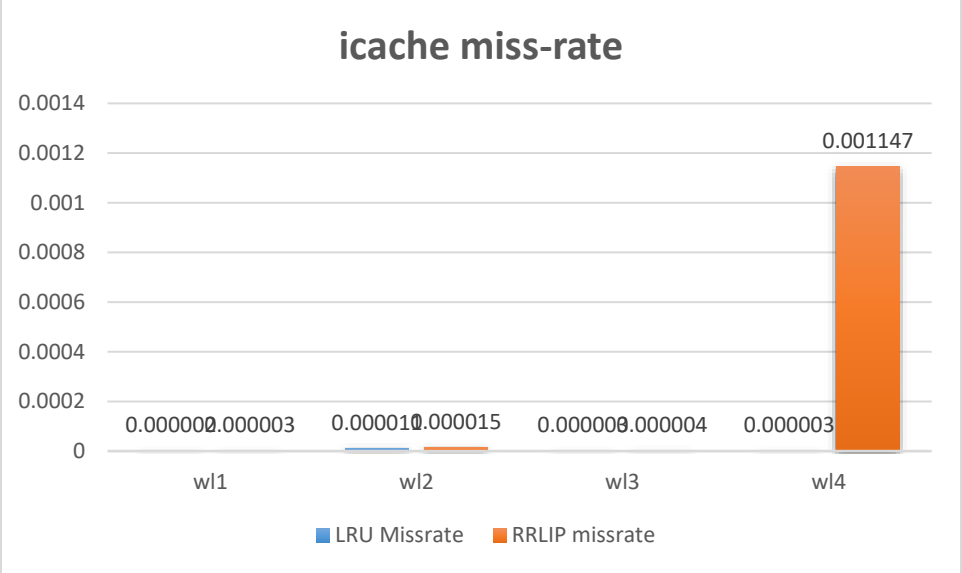

Figure 4: Icache miss-rate L1-level for LRU vs RRLIP

We can see that the policy we are using is not very well suited to the L1 caches because most of the entries in L1 cache are re-referenced in short distance. Because we are evicting entries that have not been re-referenced at least threshold times, and L1 caches have small re-reference distance we are removing entries that are going to be re-reference shorty and causing extra misses where they have to go the L2 cache. Consequently under the current policy the L1 miss-rate is expected to be higher than LRU but the main question is which factor (L1 miss-rate or L2 miss-rate) plays a bigger role in determining the better overall system performance?

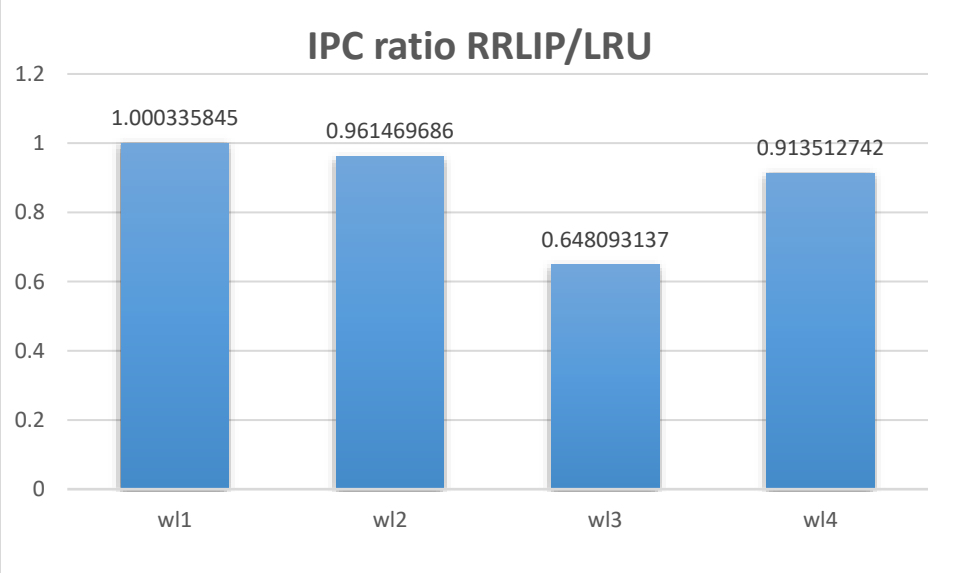

Figure 5: IPC ratio of re-reference policy and LRU policy.

From figure 5 we can see that the IPC ratio is below 1, therefore cost of L1 misses affected the overall system performance more than L2 miss-rates and a tradeoff balance is not kept to provide a better overall performance. The paper however claims that system performance is improved if we reduce the L2 miss-rate but they failed to provide the data. This could be caused by mainly three reasons; my implementation is wrong, I didn't use the author's benchmark because of license issues and the design is heavily influenced by the workload, or the authors did not check the overall IPC for their simulation. Whatever the case is I will try to contact the author and let them know of the problems I faced.
For now I used a mixed mechanism where the L1 caches use LRU and the LLC uses RRLIP(Re-reference LIP). The result I got is:

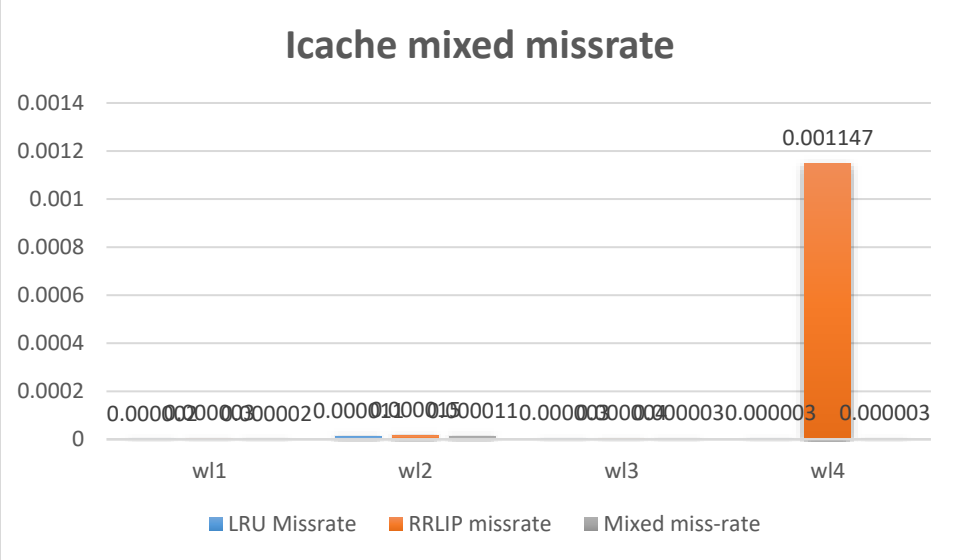

Figure 6: Icache miss-rate

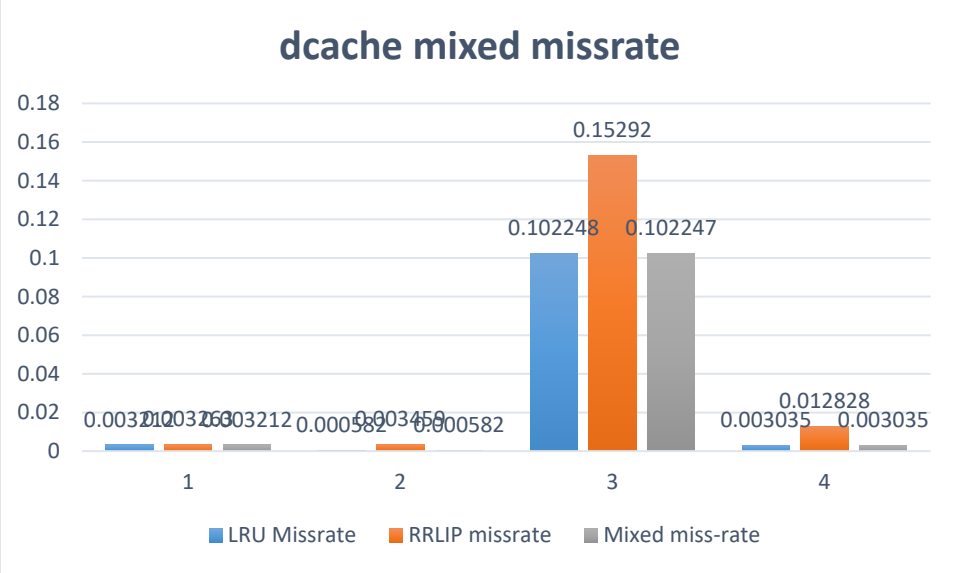

Figure 7: Dcache miss-rate

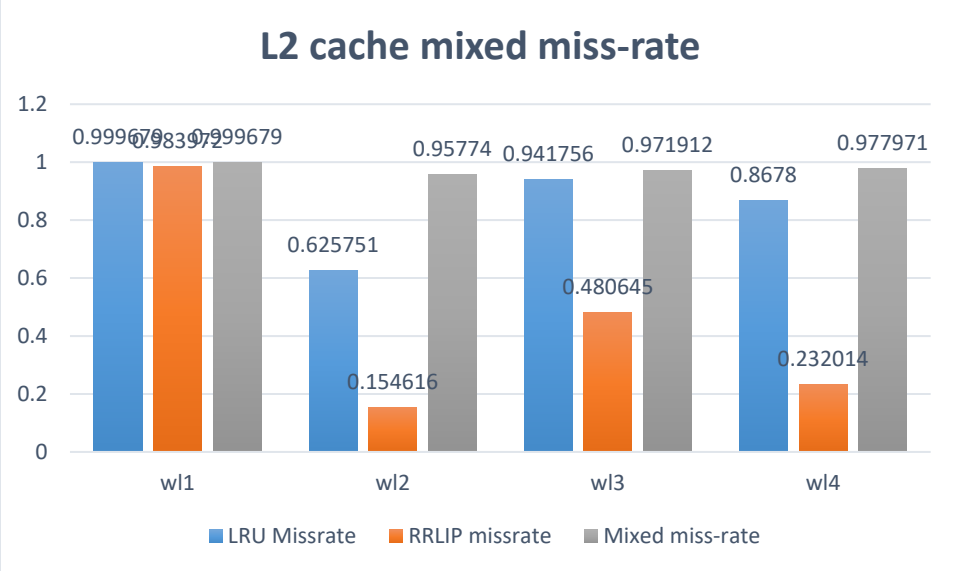

Figure 8: L2 cache miss-rate

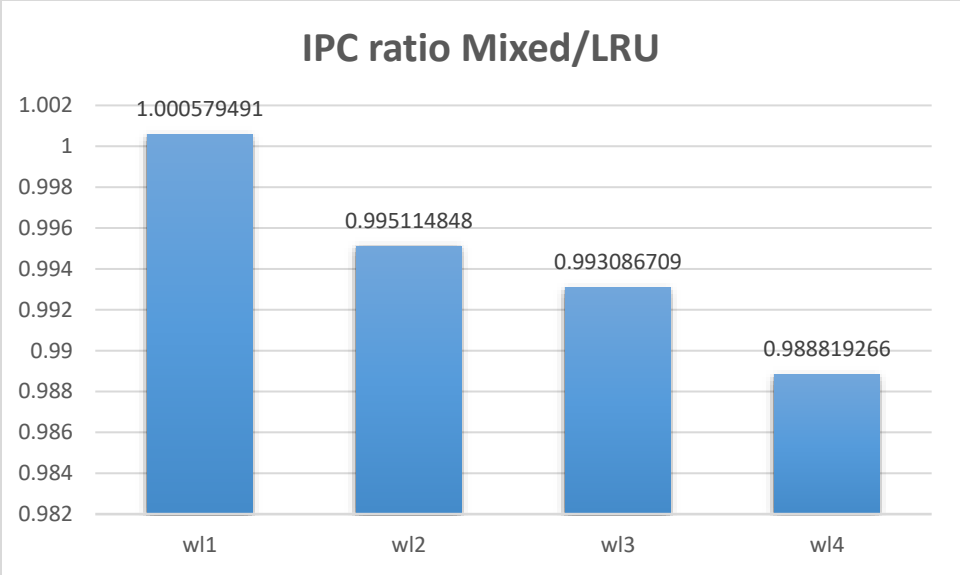

Figure 9: IPC ratio of mixed policy vs LRU policy

As we can see from the figures 6 and 7 the miss-rate of mixed and LRU are the same and we got the result we expected. But for L2 cache miss-rate is slightly higher for the mixed because the recently re-referenced entries are evicted and since the L1 caches miss are based on LRU where newly inserted entries are in the MRU, the L2 will not reflect the working set, hence misses in L1 are more likely to have been evicted in L2, and the policies do not support each other's weaknesses. All together when we see the IPC ratio it is almost 1 thus mixed mechanism cannot solve the overall system performance we faced in the full RRLIP configuration.

V. Discussion

The design and implementation of this policy has advantages as well as numerous limitations. The upsides of this design is it is able to optimize the policy to solve a major access pattern issue in Last Level Caches, which is long re-reference distances. Thus by recognizing that LLCs filter out temporal locality, it will only give entries the privilege of staying in the cache is they prove that they are re-referenced more often than other entries. Additionally, the policy distributes, the eviction decision reliability solely from the insertion policy to both the insertion and the promotion policies by realizing re-reference distance is more important in LLC than recency of access. Last but not least, the design also builds on already available LIC policy thus it is easier to implement. This gains also carry liability with them.

One of the disadvantages of the design is it fails to recognize the characteristics of upper level access and how changes in the LLC policies affect higher cache level performance. The paper derives its conclusion only based on the principle that lower LLC cache misses would translate to better overall performance and fails to address other higher level cache might sacrifice performance both depending on the policy and its interaction with LLC. Secondly, for each entry in the cache we have to add a counter and a comparator and the area cost of these additions is not taken into consideration. Lastly,

as we have seen from the result in section IV, the threshold values for the policy is highly workload dependent and the design fails to provide this flexibility and the ability to optimize the threshold value for different workloads.

In conclusion, the policy has both its upsides and downfalls. But, according to my findings the downfalls outweigh the benefits and we need to reassess the policy. My suggestion is to come up with a combination of policies for each level of the cache such that we can address issues in each cache, while keeping in mind the inter-cache policy trade-off from varied policy.

VI. References

1. *John.L.Hennessy and David.A.Patterson, "Computer Architecture: A Quantitative Approach, Fourth Edition", in Morgan Kaufmann Publishers.*
2. Sreya Sreedharan, Shimmi Asokan, "*A Cache Replacement Policy Based on Re-reference Count"*