Cache Replacement: LRU vs Protected LRU vs SCORE

Sean Koppenhafer
Portland State University
Portland, OR
Koppen2@pdx.edu

Luis Santiago
Portland State University
Portland, OR
lsantiag@pdx.edu

Cache replacement algorithms play a major role in cache performance. In this paper, we explore two new cache replacement algorithms: SCORE and Protected LRU. These replacement algorithms are twists on the classic LRU replacement policy that promise better performance for lower level caches. Both algorithms are explained in detail and then are implemented into the SimpleSim Simulator. Various simulations are run on the implemented replacement algorithms to compare their performance to LRU. Finally, the results of the simulations are presented and explanations of the results are put forward.

I. INTRODUCTION

Cache replacement algorithms play a major role in cache performance. Incorrect evictions will lead to more cache misses down the road. Choosing the best line to overwrite on every cache miss is the job of the cache replacement algorithm. In this paper, we have explored two new cache replacement algorithms that promise better cache performance.

LRU (least recently used) is the most popular cache replacement algorithm that almost anyone will know of. The SimpleSim Simulator has an implementation of LRU that ships with the simulator. In a LRU replacement algorithm, the cache line in the set that has been the least recently used will be chosen for eviction. The logic behind this is that if the line has not been used in a while, it is less likely to be needed again in the future than other lines in the set.

More modern replacement algorithms have emerged that aim to outperform LRU. The two replacement algorithms that we have chosen to examine are protected LRU (pLRU) and SCORE. Both SCORE and pLRU add additional layers of criteria onto the selection process to take into account more than just how recently a cache line has been used when choosing an evictee. This makes the implementation of the replacement algorithm a bit more complex, but the proposed performance increases are enticing.

II. PROTECTED LRU

Protected LRU (pLRU) is somewhat similar to LRU in that the least recently used cache line is still evicted. However, there is an added layer onto the process of choosing which cache line to evict: a function is ran that limits the number of lines that can be considered for LRU eviction in a set before the LRU replacement algorithm is ran. All of this is done to attempt to ensure that lines that have been accessed the most in a set will be saved regardless of how recently that cache line has been accessed.

Beyond standard LRU, pLRU takes in two parameters: max counter value and number of ways to save. The max counter value equals the max value the hit counter can reach for a cache line. The numbers of ways to save tells pLRU how many lines to remove (protect) from the set before running LRU on the remaining cache lines. Both of these parameters use will be explained in more detail in the following paragraphs.

PLRU in implementation works like this. Each cache line has a hit counter embedded in it. When a cache line is filled from main memory, the hit counter is set to 0. On each cache hit, the hit counter for that line is incremented by 1. If the hit counter value post-increment equals the max counter value, all lines in the set will have their hit counter shifted right by 1 (to divide the hit counter by 2). This ensures that a cache line cannot stay in the cache indefinitely and kill performance. The LRU value for each of the lines are also updated on a cache hit.

On a cache miss, pLRU first generates the subset of unprotected cache lines that the LRU algorithm uses to pick a line to evict from. The number of cache lines to protect is determined by the ways to save parameter mentioned above. For example, let's say N = ways to save. The N number of lines with the largest hit counter values will be protected and therefore will be removed from the pool of cache lines that the LRU algorithm can choose to evict from.

After the "ways to save" number of cache lines have been protected, LRU is ran on the remaining cache lines to choose which line to evict. Through this two step process, pLRU ensures that cache lines that were accessed the most in the past but are currently not the most recently accessed stay in the cache. Which can lead to performance gains if the time between accesses on heavily used cache lines are longer than would be feasible in a pure LRU system.

III. Score Cache Replacement

SCORE uses a per line score system to represent the access behavior of each cache line. To do so it uses initial scores, increase and decrease scores, threshold values and two methods of evicting cache lines. Next, we provide an overall description of SCORE behavior. Followed by a more detailed explanation of how the different components work.

SCORE works as follows. When a line is brought from memory an initial score is set for the line. Based on how the line and the set are accessed, the initial score is increased or decreased on a per-line basis to reflect the past behavior of the set. When eviction is necessary, the score for all lines in the set are compared and one of two eviction methods is used to determine which line to evict.

The first eviction method involves evicting the line with the lowest score. If there is no line below the eviction threshold score, a line with the lowest score is evicted. This has a similar behavior to LRU. The other eviction method is random. The random policy is applied by comparing all lines under the eviction score threshold and selection of them at random to be evicted.

The initial score affects when a line may fall below the threshold value and be eligible for eviction. A higher initial score is beneficial for programs with high locality and a lower initial score might be better for programs with low locality. The designers of SCORE suggest that the specific value of initial score should vary not only according to different applications but also different phases of the program.

The value of initial score changes with the objective to make the cache more effective. After a pre-set number of cache accesses, the cache misses are compared and the initial score is either incremented and decrement by a score step. The direction of the steps is determined by monitoring the cache misses on an interval of cache access. After each interval, the number of misses is compared with the number of misses of the previous interval. If the misses of current interval are less than the ones for the previous interval the initial score step changes on the same direction otherwise we change the direction of the steps.

After the initial score is set, each value is incremented or decremented to reflect the past access of each line. When a line is a hit that line is increased by the increased velocity and all other lines on the set are decreased by the decreased velocity. Also, when a miss or evictions happens on a set all the other lines are decreased by the decreased velocity. The decreased and increase velocity values are fixed values. To avoid the scores from going to low, the increase velocity is higher than the decrease velocity.

IV. Simulation Methodology

Initially, we needed to find a baseline L2 cache configuration to work with that had a large miss rate when using LRU replacement. To make this more realistic, the main memory access time in cycles was increased from 18 to 150 cycles. Doing so allowed for a more dramatic IPC decrease when many misses were happening in the L2 cache. Something that may have been masked if the memory access time were left at 18 cycles.

To find our baseline cache configuration, we ran sweeping configuration tests on LRU, looking for the knee of the curve where the miss rate began to increase rapidly. The number of sets in the cache, the size in bytes of each of the lines, and the associativity of the cache were changed in this simulation. Ultimately, a L2 cache with 32 sets or less and 64 byte lines ultimately ended up being the cache configurations that had rapidly increasing miss rates – as shown in the plot below.

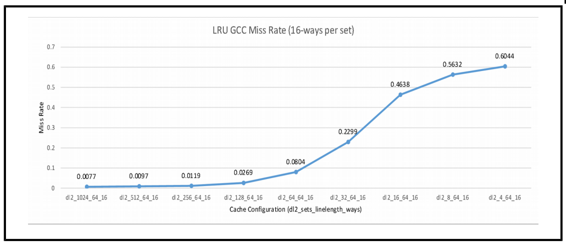


Fig 1 – LRU Miss Rate vs Different Cache Configs

Once a baseline set of cache configurations were found on LRU, we ran the same cache configurations on pLRU and SCORE. Again, the miss rate for the L2 cache was the focus of the simulation runs. Because both SCORE and pLRU have additional parameters that LRU does not have, more runs were done on SCORE and pLRU at each cache configuration than was done on LRU. In each additional run, the SCORE/pLRU parameters we changed to attempt to produce better L2 cache performance.

An additional piece of data was requested to be collected during the protect demonstration. The data that was requested was the number of cycles between the last time the cache line was hit and when the cache line was evicted. Knowing the delta between the cache line access and cache line eviction would provide with additional data that would allow us to evaluate the performance of each of the algorithms.

V. Results

In general, the results from our simulations showed that pLRU and SCORE had worse performance than LRU. The bar graphs below show the miss rates for the various cache replacement algorithms on a single cache configuration. Additionally, all benchmarks included in the simulator were ran on each replacement algorithm to fully explore the performance of each of them.

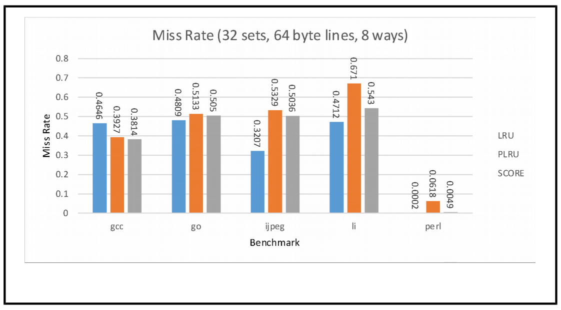


Fig 2 -Miss Rate (32 sets, 64 byte lines, 8 ways)

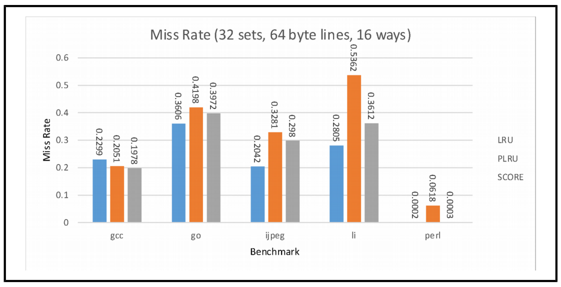


Fig 3- Miss Rate (32 sets, 64 byte lines, 16 Ways)

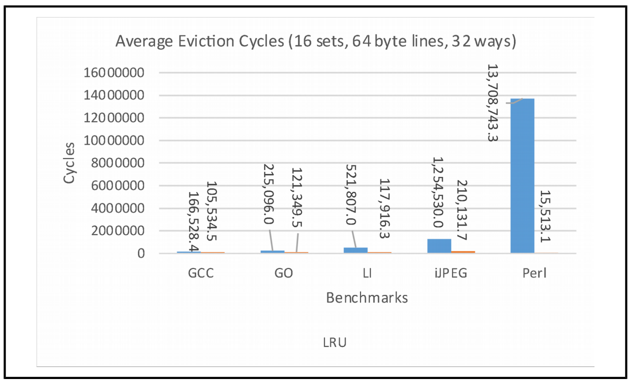


Fig 4 (16 sets, 64 byte lines, 32 ways)

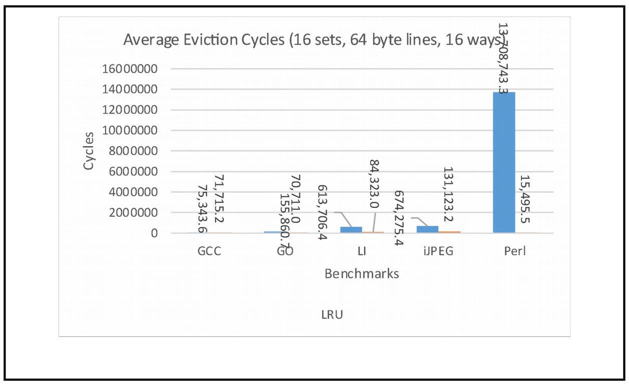


Fig 5 (16 sets, 64 bye lines, 16 ways)

A few possible explanations for this decrease in performance have been explored. While reading through the SCORE paper, we noted that the authors intended to use the SCORE replacement algorithm on a unified L3 cache. Because the SCORE paper was written in 2014, we concluded that the L3 cache hierarchy the SCORE authors were envisioning was a part of a muli-core system. That would mean that there would be multiple L2 caches feeding misses into the L3 cache. From other papers we have read in the course, we know that multi-core systems place more demands on the caching hierarchy than a single core system does. We also know that multi-core systems introduce coherency misses – something that does not exist in a single core system.

In our experiment, the SimpleSim Simulator was written in the 90s. The SimpleSim Simulator is a single core simulator with the L2 cache as the lowest level of caching hierarchy. Above the L2 cache, there is only a L1 instruction cache and a L1 data cache. Because the SimpleSim Simulator is a lower complexity system than the systems the authors were considering in the SCORE paper, it is possible that the performance gains with pLRU and SCORE are only seen in more complex multi-core systems. It is not hard to imagine that having coherency misses and more memory accesses due to multiple cores executing would favor different replacement algorithms than single core systems that lack these extra demands.

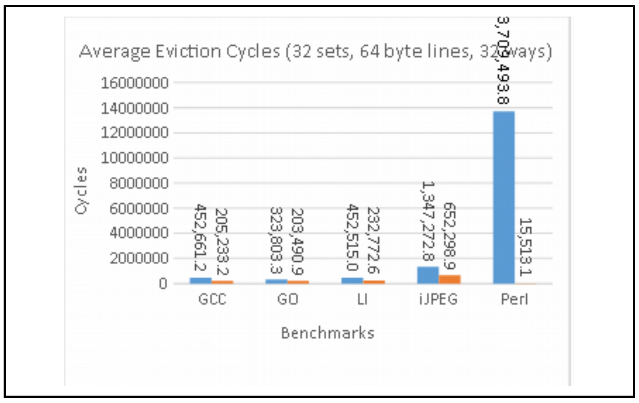
Another possible explanation had to do with the benchmarks that were used to test the performance of the replacement algorithms in our simulations versus the simulations done by the original authors. In SCORE and pLRU, Specman06 was used as the simulation benchmark. For our simulations, we used the built in benchmarks that shipped with SimpleSim. These benchmarks included gcc, perl, ljpreg, and Specman95. Many of these benchmarks had very small working sets in comparison to the working set size present in Specman06.

For example, the Perl benchmark had a very small working set. Almost all of its cache misses were compulsory misses. Most of the data was only used once before being discarded and never used again. On this benchmark, LRU did very well while pLRU and SCORE did very poorly in comparison. LRU was better because the Perl benchmark is tuned to work well with a replacement algorithm that only evicts based on what the oldest cache line in the set is.

Of all of the benchmarks in SimpleSim, gcc was the only benchmark that consistently showed better results on SCORE or pLRU vs LRU. Gcc had the largest working set of all of the benchmarks in SimpleSim and also repeatedly referenced different sets of data. This lead us to believe that gcc is the most accurate benchmark to use when comparing the results that the authors of the papers got using Specman06.

On presentation night, Professor Zeshan brought up another possible explanation for the performance results. If the cache lines evicted in pLRU and SCORE were staying in the cache longer than the evicted lines in LRU, this could also explain the performance decrease. Because both SCORE and pLRU have more complex rules for which cache line to evict, it is more likely that a cache line that is not being used will stay in the cache longer than it needs to be. In LRU, it is likely that these unused cache lines will be evicted more quickly because the rules for eviction in LRU are simpler.

To test this theory, we modified the SimpleSim simulator to calculate the delta between the last cycle that a cache line was hit and the cycle that the cache line was evicted. In interest of time, we only implemented this feature on LRU and pLRU. An additional variable was added to each cache line that stored the simulation cycle number on each cache hit. Then, when the cache line was evicted, the delta between the current simulation cycle and the last hit cycle stored in the cache line was printed to a logfile. Data produced from that experiment is shown below.



Average Eviction Cycles (32 sets, 64 byte lines, 32 Ways)

Given the results, it appears that it is true, in general, that cache lines in pLRU stay in the cache for longer periods of time before being evicted than in LRU. SCORE most likely also suffers from the same issue since SCORE saw a similar performance degradation in initial testing.

VI. Conclusion

From the simulations that we ran for this paper, pLRU and SCORE would appear to be inferior cache replacement algorithms when compared to LRU. However, the promising results from the gcc test tell a potentially different story. In benchmarks that have larger working sets, it appears that pLRU and SCORE have a slight upper hand. While it is impossible to know without running similar simulations on a more modern simulator, we are convinced that pLRU and SCORE would outperform LRU for a unified L3 cache in a multi-core system running a complex benchmark like Specman06.