**Cache Replacement: LRU vs Protected LRU vs SCORE**

**Sean Koppenhafer and Luis Santiago**

**Abstract**

Add abstract

**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 Simple Sim Simulator has an implementation of LRU that ships with the simulator. LRU performs LRU evicts the line in the set that has been least recently used. LRU makes the assumption that if the line has not been touched recently, then it is least likely to be touched again in the future.

More modern replacement algorithms have emerged that aim to outperform LRU by using more than just how recently a cache line has been used. The two replacement algorithms that we have chosen to examine are protected LRU and SCORE. Both of which will be explained later in greater detail.

**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: 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 when the last time they were accessed was.

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 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). 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 can pick 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 removed from the pool of cache lines that LRU can evict.

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. pLRU ensures that cache lines that are accessed the most in the past but are currently not the most recently accessed stay in the cache. Which can lead to potentially lead to gains in performance.

**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. Based on how the line and the set are access, the initial score is increased or decreased on a per-line basis to reflect the past behavior of the set. When a line needs to be evicted from the set, the score for all the lines on the set are compared and one of two eviction methods is used.

The first eviction method is the lowest score, if there is no line below the threshold 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 to lines under threshold, choosing the line with the lowest score is not always the best choice [2].

The initial score affects when a line may fall below the threshold value and be eligible for eviction. A higher initial value 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 “The specific value of initial score should vary not only according to different applications but also different phases of the program.” [2]

The value of initial score is changes in an attempts to make the cache more effective. After a preset number of cache accesses (interval) the cache misses are compared and the initial score is either incremented and decrement by a score step. The direction of the steps is determine 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 [2].

**Simulation Methodology**

Initially, we needed to find a baseline L2 cache configuration to work with that had a large miss rate with LRU. 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.

To find this 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. 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 seen in the plot below.

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.

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 hit and eviction would show us how long each cache line was staying in the cache between the different algorithms.