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

**Sean Koppenhafer an Luis Santiago**

**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, lets 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.

**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 10 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.