Implementation of SCORE and adaptive cache replacement policies.

Andrew Capatina, Shaun Crippen, Bhargavi Raviprakash, Michael Zarubin

Abstract:

SCORE, LFU, and ADAPTIVE cache replacement policies were implemented using the SuperScalar simulator. LFU was added to the simulator to test the ADAPTIVE policy implemented by [2]. The purpose of the research was to validate the results found in [1] and [2]. SCORE was proposed as a better alternative to popular cache policies like LRU. We decided to implement SCORE since it had a simplistic algorithm which allowed us to thoroughly examine the results provided by [1]. ADAPTIVE was chosen since we were curious how a policy that used two other cache algorithms could provide enough benefit from decreasing miss rate was also incurring a large CPI penalty. It seemed that on paper a cache algorithm that used two separate caches could not outperform a cache of twice the size with a single policy.

Test results showed that SCORE suffers from thrashing in caches with low associativity and that ADAPTIVE has less misses when compared to LRU but at the cost of execution time, which is congruent with the previous research. The results showed that SCORE performed equally to LRU in the GCC benchmark. This was due to SCORE being implemented as a static policy as opposed to the dynamic SCORE implemented in [1]. SCORE was run with varying cache sizes to improve both thrashing and to bring SCORE's performance closer to the dynamic SCORE policy implemented in [1].

Introduction:

As processor performance improves memory improvements tend to not follow. In the past memory improvements were introduced at a slower rate compared to processors. The result is reduction of Average Memory Access Time (AMAT) is a priority. For example, if this is not addressed hardware resources in the instruction fetch stage could stall which would then affect the rest of the pipeline. To decrease AMAT cache miss rate must decrease. Current methods to reduce the miss rate include increasing the size of the cache, associativity, adding a victim cache, pre-fetching, and software optimizations [3]. Another method is trying a new cache replacement policy, as will be discussed in this paper. Most replacement policies enforce temporal locality by keeping blocks which are most likely to be used. The most common cache replacement policy is LRU [4]. This paper will compare the SCORE and adaptive replacement policies with LRU. The SCORE replacement policy uses a score for each block which gives indication of the chance of the block being used again. The adaptive replacement algorithm uses a voting method to choose between competing replacement policies. In terms of power consumption and area usage, the adaptive policy uses much more resources than LRU and SCORE. This tradeoff of the design may deter designers from using adaptive. For testing, it is obvious that the associativity must be greater than 1 for the cache replacement policy to take effect; otherwise the cache is direct-mapped.

The SCORE Replacement Policy:

High Level Overview:

The SCORE replacement policy associates a score value with each item brought into the cache. This value is added or decremented based on a cache hit or miss. A threshold value is used for selecting cache blocks to evict. The replacement algorithm will be discussed below.

The SCORE value:

The initial score can be changed depending on the workload of the program for better performance. For low locality use a low initial score because there is less of a chance a hit will occur so cache blocks will need to be replaced more often. For high locality, use a high initial score so that other blocks missed in the set don’t have their scores drop to the minimum value so quickly. [1] uses a dynamic initial score and high score based on the associativity. Let n be the associativity, the max score is equal to (2^n - 1). The score value determines how likely a block will be accessed in the future. Therefore, the replacement policy will reference this value.

Handling Cache Hits and Misses:

For a cache hit, one cache block’s score value is incremented while the rest are decremented; this occurs within the same set only. The cache block that hit must have the score incremented. For a cache miss, the replacement policy must take effect. Figure 1 shows the replacement algorithm. The replacement algorithm begins by searching the cache looking for all blocks under the threshold value. If there were blocks found below the threshold, they are all candidates for eviction. The replacement is then randomly chosen between the blocks below threshold. If there were no blocks below the threshold, choose the block with the lowest score. Keep in mind the replacement policy choices are done within the cache set.

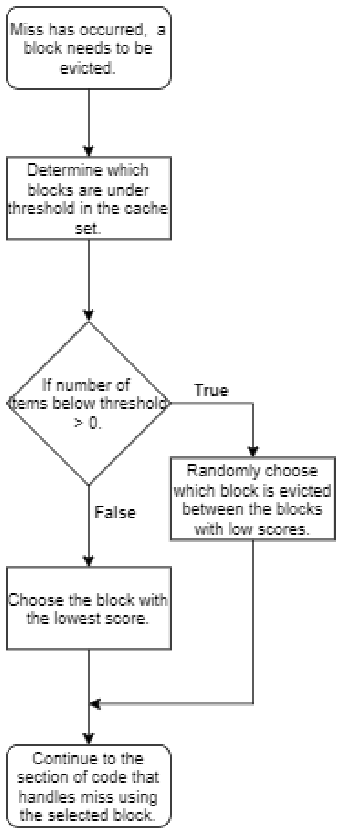

Miss has occurred, a block needs to be evicted.
Determine which blocks are under threshold in the cache set.
If number of items below threshold > 0.
True
Randomly choose which block is evicted between the blocks with low scores.
False
Choose the block with the lowest score.
Continue to the section of code that handles miss using the selected block.

Figure 1: High level block diagram for SCORE replacement policy.

Adaptive Replacement Policy:

High Level Overview:

The adaptive cache is by far the most resource demanding replacement policy mentioned in this paper. The adaptive cache essentially uses a voting system to decide which replacement policy should be used on any given miss. To further explain the policy, it is best to begin by describing the data structures used followed by the replacement policy algorithm.

Data Structures:

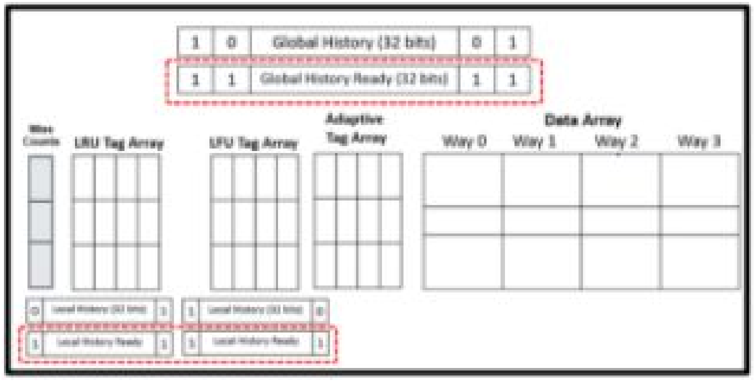


Figure 2: The data structures that need to be defined for the adaptive replacement policy.

Figure 2 shows the structures used for the adaptive replacement policy. The only data structure not used was the Miss Counts array because the voting algorithm doesn't require the information and [2] didn't clearly explain the intended purpose for this structure. The LRU tag array and LFU tag array contains the tags of the instructions/data brought into the cache and not the data associated with it. The LRU/LFU tag arrays have their own replacement policy and it never changes. This is so that the history can be recorded for each policy and a decision can be made for the Adaptive tag array. The Adaptive tag array's replacement policy is dependent upon the history information collected. The local history buffers for the LRU and LFU tag array contain the history bits for the result of the cache access; shift a 0 for a cache hit and 1 for a miss. The global history buffer stores the policy used for the adaptive cache. Since only 2 policies are being used, 0 and 1 can represent the two policies. The data array is not associated with the LRU and LFU tag arrays. The data array is updated when the adaptive tag array is updated. Lastly, the local and global history READY buffers are used to indicate when the local and global history buffers are full. Each time the local/global history buffer is updated a 1 is shifted into these buffers until they are full. The READY buffers are the red outlined structures in Figure 2.

Cache Block Replacement Policy:

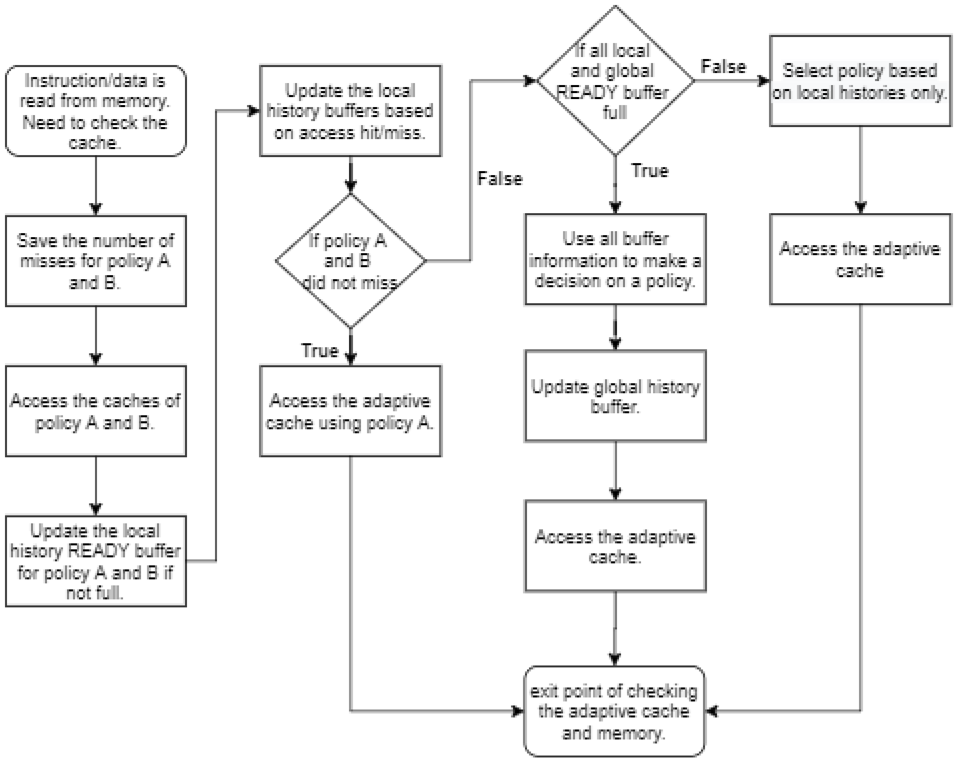

Instruction/data is read from memory. Need to check the cache.
Save the number of misses for policy A and B.
Access the caches of policy A and B.
Update the local history READY buffer for policy A and B if not full.
Update the local history buffers based on access hit/miss.
If policy A and B did not miss
True
False
Access the adaptive cache using policy A.
If all local and global READY buffer full
False
True
Use all buffer information to make a decision on a policy.
Update global history buffer.
Access the adaptive cache.
Select policy based on local histories only.
Access the adaptive cache
exit point of checking the adaptive cache and memory.

Figure 3: Adaptive cache replacement high level block diagram.

Figure 3 shows a high level block diagram of the Adaptive cache replacement policy. When an instruction or data is read from memory the caches for policy A and B (LRU and LFU) must be accessed first. Prior to access the number of misses should be recorded so that it can be known if a miss has occurred. Once the caches A/B are accessed the local history buffers and local history READY buffers are updated for each. If no misses have occurred just use policy A for the adaptive cache. If a miss has occurred then it must be determined if the READY buffers are full. If all READY buffers are full then use a voting algorithm with the local and global history buffer values as input. A simple voting algorithm could be to check if policy A is less than B indicating more cache hits. If the ready buffers aren’t full, then use the local histories only for voting.

Information Needed For Implementation

SCORE Policy

The score value was added to the cache_blk_t struct definition of cache.h. Furthermore, the cache_t struct contains the max, initial, and threshold values (cache.h). The variables were initialized in cache_create() function of cache.c. Lastly, the replacement algorithm was added in the cache_access function of cache.c.

Adaptive Policy

The LFU replacement policy had to be added for this method. First, the LFU frequency counter was added in the cache_blk_t definition of cache.h. This variable is initialized in cache_create() function of cache.c. For the replacement policy of the LFU method, a counter is used for each block. But when there are equal frequencies, the waychain is ordered with the least recently used item being at the tail of the chain. This solves equal frequencies in the cache set.

The adaptive data structures were implemented in sim-outorder.c. The data structures were declared and initialized in that file. The access/voting algorithm shown was implemented wherever the il1, il2, dl1, dl2, caches were present. The prior data structures for the caches were preserved and if statements were added checking if the adaptive cache pointer or the regular cache pointer was null. If we were to re approach this problem, it would be best to define the adaptive data structures in a struct and use a function returning the cache pointer for the access algorithm. This would make the code more readable and maintainable.

Simulation Environment and Testing Implementation:

Simulation:

The simulator used was created by Dr. Zeshan Chrishti at Portland State University. The code can be obtained from the following link: http://www.cecs.pdx.edu/~zeshan/ss.tar.gz

The simulator was run from a machine using a linux operating system. The benchmarks we used were: GCC, LI, PERL, GO.

Testing methodology

We tried to independently validate the results given by [1], and [2]. The SCORE policy paper [1] claimed that on average IPC was 4.9% higher than the LRU policy. They used multiple SPEC CPU2006 benchmarks including: 401.BZZIP2, 403.GCC, 429.MCF, and 436.cactusADM. While we used different benchmarks, we expected that the IPC should still increase, and our results show that percent improvement in IPC for SCORE was lower than or tied with the LRU policy. The reasons for this are discussed in the results section.

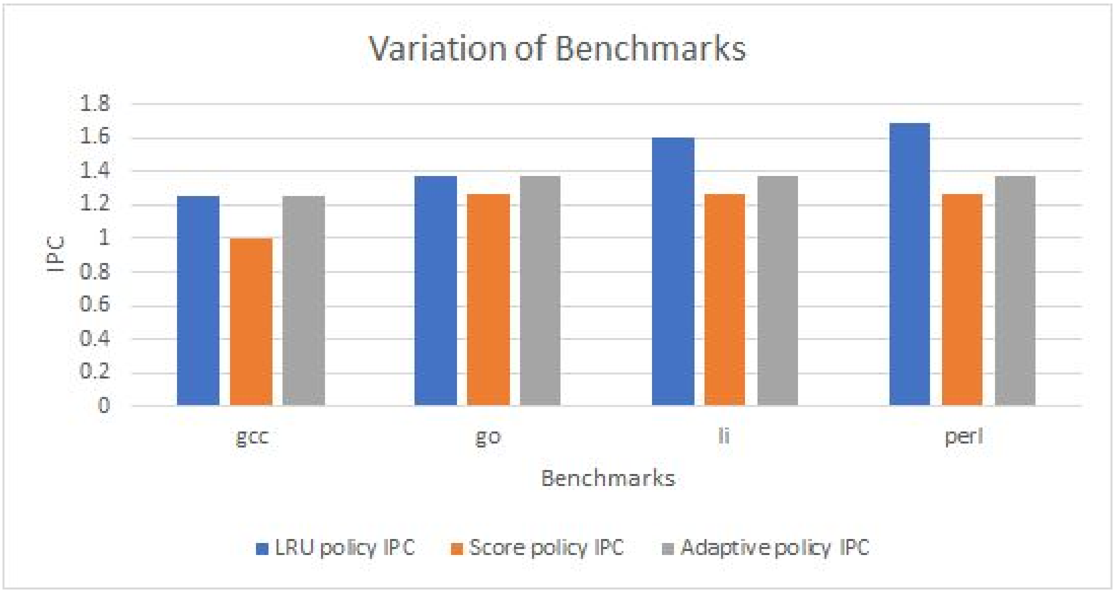
For [2], they argued between the LRU and LFU policies, the LRU is better. So that when LFU and LRU are used as the only policies in the adaptive replacement strategy the number of misses decreases but has a higher CPI. Its execution time is slower, owing to the increased overhead of tracking two policies and then comparing them, and selecting the winner. To validate those results we specifically measured the miss rate (in MPKI) for [2] between the LFU and LRU policies. Our results showed that there was very little difference between the adaptive and LRU policies, which is congruent with results produced by [2]. This is analyzed in the results section.

Results:

Variation of benchmarks (IPC)	
Cache size	32KB
Number of sets	1024
Cache block size	32B
Number of ways	2-way

The following graph shows the IPC of each benchmark for LRU, SCORE, and Adaptive cache replacement policies. As expected, the LRU is the clear winner here, with the adaptive policy matching it at best. This makes sense as the adaptive policy chose between LRU and LFU, which incurred overhead and as a result the IPC was lower than that of the LRU. The SCORE policy performed abysmally in IPC. One explanation for SCORE’s lower IPC is that in a 2-way associative cache there is a large amount of cache thrashing. Suppose a hit occurred, each block in a set that wasn’t the block that hit has its score decremented. Thus for a cache with relatively few sets, and ways evictions will occur often, this is cache thrashing where a block that may be needed again soon is evicted prematurely due to the score algorithm. Since misses occur often in caches with low associativity, thrashing is a big problem for SCORE as it is poorly optimized. Clearly it needs to be dynamic, or at least tuned in order to be considered for further use as a replacement policy.

In order to test this SCORE thrashing hypothesis, we kept the cache size constant at 32KB and varied the number of sets and the associativity, in the following graphs to see if an increased cache size led to better results. According to [1] they first implemented the SCORE policy and then took it a step further. Several math intensive algorithms were used to dynamically create and change the initial cache block score, the threshold, increase velocity, and decrease velocity of each set. After a calibration run, the dynamic SCORE policy had a 4.9% IPC lead over the LRU. Since our project did not use the dynamic aspect of the SCORE policy it is expected that the LRU should prevail in IPC. Similarly with the adaptive since the LRU policy was one option to be used, its IPC is also higher than that of score.



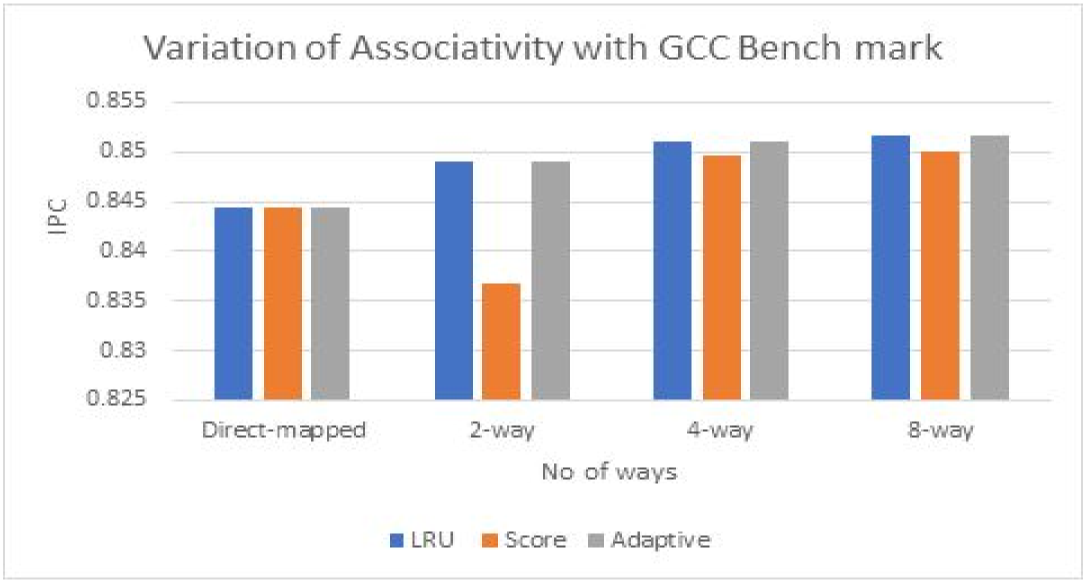
Varying cache associativity compared to miss rate

Variation of Associativity with GCC Benchmark	
Cache size	32KB
Number of sets	1024, 512, 256, 128
Cache block size	32B
Associativity	Direct-mapped, 2-way, 4-way, 8-way

For a direct mapped cache, all three policies had equivalent performance. Since each cache block would be evicted on a miss, it is expected that policy had to affect. The data has only one place it could be, and so a miss always incurs an eviction, or a call to main memory (HDD, SSD). Therefore, all cache policies should have the same IPC.

As the associativity increased, so did the IPC of SCORE. Since SCORE lacked the dynamic aspect of [1], the thrashing significantly throttled IPC for low associativity caches. There is a small increase from 4-way to 8-way as such we did not need to run additional associativity tests. The cache size was constant for each trial, as was the number of instructions executed from the GCC benchmark, and the CPU clock speed. As such, the increase in IPC is due to less cache thrashing. Since LRU and adaptive are better optimized, they experience less thrashing at all associativity levels. SCORE needs a lot of tuning to make it work for each benchmark, and our algorithm lacks the dynamic adjusting of the threshold, initial score, increase, and decrease velocities necessary to see an IPC improvement over LRU.

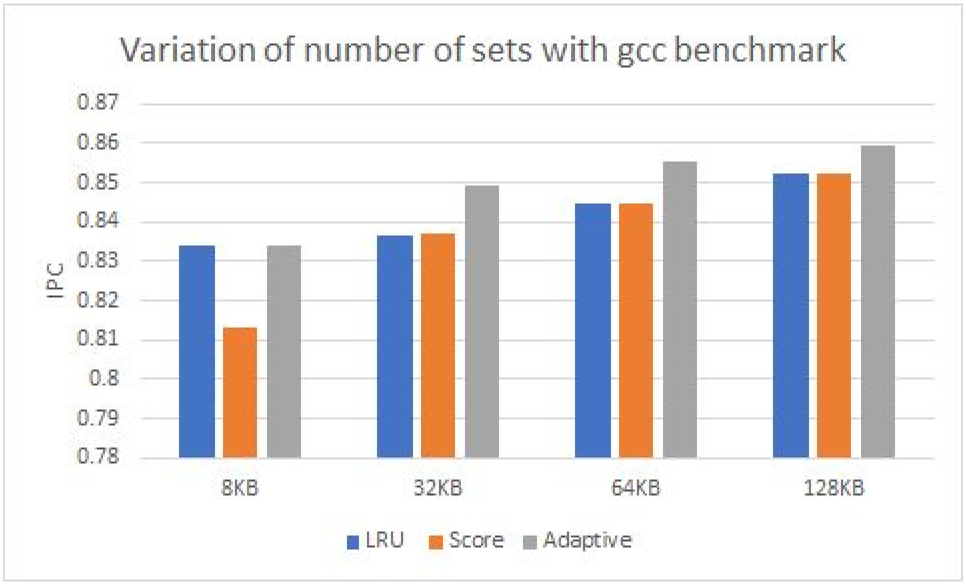
Finally, once thrashing is mitigated by using a more associative cache, the IPC of SCORE, LRU, and ADAPTIVE are roughly equal. They vary by less than a tenth of a percent. Since [1] had at best a 4.9% increase over LRU, and our results were roughly equal without the dynamic aspect in a highly associative cache, The results of [1] are consistent with results. We only tested this on the GCC benchmark, and further analysis is needed to determine if SCORE can compete with LRU consistently across other test suites without the dynamic portion.



Varying number of cache sets compared to IPC

Variation of number of sets with GCC benchmark (IPC)	
Cache size	8KB, 32KB, 64KB, 128KB
Number of sets	128, 512, 1024, 2048
Cache block size	32B
Associativity	2-way

To double check that thrashing was the cause of a low IPC for SCORE as compared to LRU and ADAPTIVE we ran the GCC benchmark with different cache sizes. Since the larger a cache is the less data conflicts there will be (thrashing). As the cache size increases we see that the IPC for SCORE goes up as expected. At a cache size of 32KB and up, the score performs equivalent to LRU, this matches the results from [1], where again the lead that SCORE had over LRU is attributed to the dynamic improvements made to SCORE.

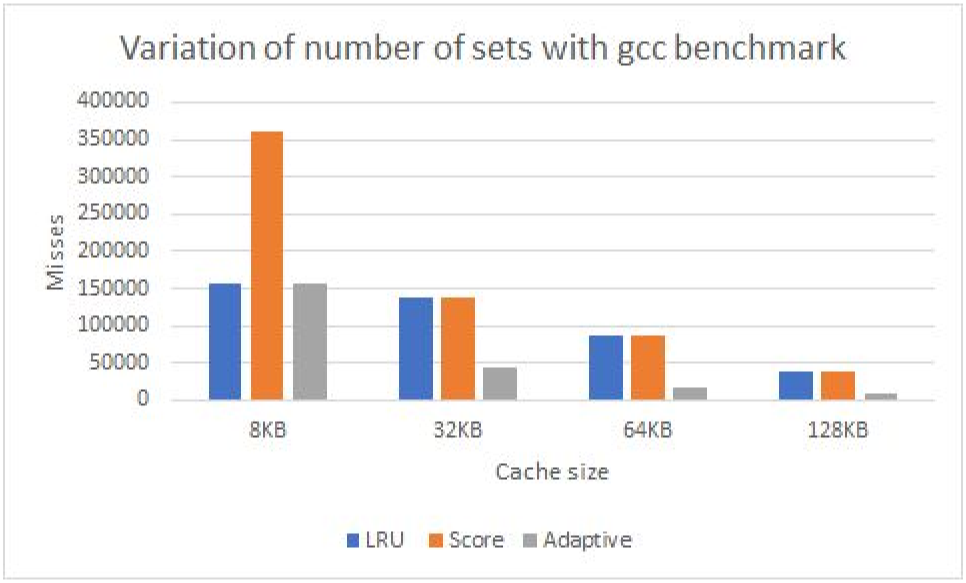


Varying number of cache sets compared to miss rate

Variation of number of sets with GCC benchmark (miss rate)	
Cache size	8KB, 32KB, 64KB, 128KB
Number of sets	128, 512, 1024, 2048
Cache block size	32B
Associativity	2-way

The number of misses for SCORE at 8KB is astronomical. Which is expected from the last graph, where the 8KB cache had a low IPC. This shows that it was due to many misses occurring causing thrashing.

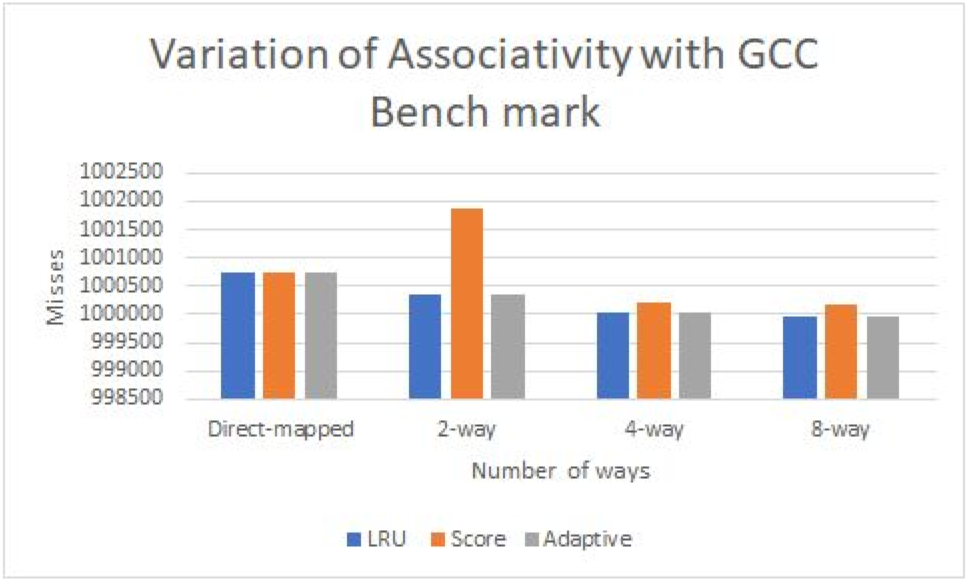
The miss rate for the ADAPTIVE , and LRU policies are the same for a small cache (small caches are easy to get misses in), however as cache size increases so does the number of misses for the adaptive policy. According to [2] this is the expected behavior, at the cost of a slightly higher execution time. Our results are consistent with those from [2].



Varying cache associativity compared to miss rate

Variation of Associativity with GCC benchmark (Miss rate)	
Cache size	32KB
Number of sets	1024, 512, 256, 128
Cache block size	32B
Associativity	Direct-mapped, 2-way, 4-way, 8-way

As expected then, the number of misses for a set cache size decreases with increasing associativity. This again shows that SCORE suffers from excessive thrashing with low associativity, ignoring direct-mapped architecture.

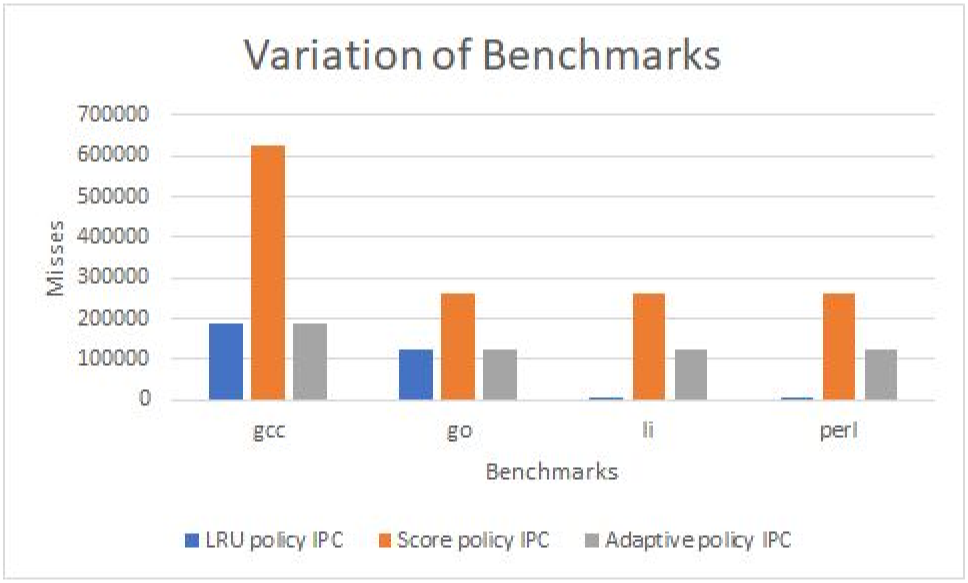


Miss rate across benchmarks

Variation of benchmarks (miss rate)	
Cache size	32KB
Number of sets	1024
Cache block size	32
Associativity	2-way

For a set cache size we compared the miss rates of all three policies to check if the ADAPTIVE policy had less misses than LRU. However, this trend is reversed for the Li and Perl benchmarks. We do not know much about these testing suites so we can only assume that the simulated workload was significantly different from GCC and GO.

The adaptive policy did meet expectations for the GCC suite, especially as cache size increased. The weird part is that since ADAPTIVE chose between LRU and LFU it was expected that LRU and ADAPTIVE would go toe-to-toe in number of misses. More research into the simulated workload for the benchmarks is needed in order to understand why this reverse trend is seen.



Conclusion:

We implemented both non-dynamic SCORE and ADAPTIVE cache replacement policies to validate the results of [1] and [2]. The cache policies were simulated with the SuperScalar simulator provided by Dr. Zeshan Chishti. Since the research done by [2] implemented an adaptive cache policy that used both LRU and LFU, we also added the LFU policy to the simulator.

From our research, we have concluded that SCORE, after thrashing was removed as a factor, performed equivalent to LRU. The 4.9% improvement in performance seen in [1] was due to the dynamic aspect of the SCORE policy implemented. This is where the threshold, initial score, increase, and decrease velocities are changed as the program runs. In our testing, thrashing in SCORE was removed by increasing cache size and using a chase with a highly associative cache architecture.

The adaptive policy implemented by [2] when compared to LRU, had less misses but at the cost of a slight increase in execution time. Since instruction count and clock speed are constant regardless of policy, execution time is then simply the CPI, matching the results from [2]. This can be seen in the first graph in Results, where the IPC is lower for adaptive than LRU. Graph 4 from the Results shows that the miss rate of ADAPTIVE is less than or equal to the miss rate of LRU regardless of cache size. This is also congruent with the results in [2].

References:

[1] Nam Doung, Roadario Cammarota, Dali Zhao, Taesu Kim, Alex Veidenbaum. SCORE: A Score-Based Memory Cache Replacement Policy

[2] Awrad Mohammed Ali, Stacy Gramajo, Neslisah Torosdagli. Adaptive Cache Replacement Policy

[3] Zeshan Chishti. Portland State University. Cache Performance Lecture 10 ECE585.

[4] Zeshan Chishti. Portland State University. Cache Replacement Policies Lecture 9 ECE585.