SCORE: A Score-Based Memory Cache Replacement Policy

Nam Duong, Rosario Cammarota, Dali Zhao, Taesu Kim, Alex Veidenbaum

SCORE: A Score-Based Memory Cache Replacement Policy

Nam Duong, Rosario Cammarota, Dali Zhao, Taesu Kim, Alex Veidenbaum
Department of Computer Science
University of California, Irvine
{nlduong, rosario.c, daliz, tkim15, alexv}@ics.uci.edu

Abstract

We propose SCORE, a novel adaptive cache replacement policy, which uses a score system to select a cache line to replace. Results show that SCORE offers low overall miss rates on SPEC CPU2006 benchmarks, and provides an average IPC that is 4.9% higher than LRU and 7.4% higher than LIP.

1 Introduction

Prior research has shown that a cache replacement policy can have a significant impact on performance [1]. Unfortunately, the implementable replacement policies, such as LRU, MRU or LIP [12], work well in some applications and poorly in others. This can be seen in the results of Figure 1 for the miss rates of the Last Level Cache (LLC) for a subset of SPEC CPU2006 benchmarks[1] [6].

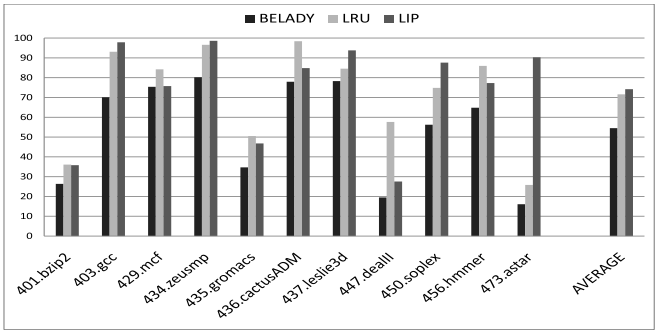

Figure 1. Miss rate of different policies.

Let us compare the Belady's algorithm[2] and the LRU and LIP policies. The main difference between LRU and LIP is the *initial position* of the new cache line in the stack. LRU puts the new line in the MRU position, while LIP puts the new line in the LRU position. After the a hit, both policies bring that the hit line to the MRU position. The results indicate that putting a new line in the MRU position works for applications with high locality, while putting the new line in the LRU position works better for applications which access a cache line mostly once. Dynamic policies proposed in [12] choose between the LRU and MRU placement dynamically which improves performance. However, there is still room for improvement as seen in the result of Belady's algorithm.

This paper proposes a Score-Based Replacement Policy (SCORE), which uses a score system to choose the victim lines. Each cache line is assigned a score, which reflects the access behavior of the line in relation to other lines in the same cache set. A line brought from memory is assigned an *initial score*. On a line hit its score is increased, otherwise, it is decreased. The scores basically predict the future: lines with low scores are less likely to be accessed than those with higher scores.

Other techniques, e.g. LRU, MRU, or LIP, can be implemented using the score system by changing its parameters. SCORE can also be made more adaptive to the application behavior by dynamically changing its parameters. Our results show that SCORE achieves low miss rates compared to previous replacement policies. It is also more stable, i.e. it achieves a much lower standard deviation in miss rates and speedup.

In this paper, we study our SCORE for the last-level cache. We present the results for a 16-way cache with the size of 1MB. Please note, however, our policy can also be applied to other levels of cache.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the SCORE policy and shows that other replacement policies can be represented under the scoring system. The experimental methodology is presented in Section 4 and Section 5 presents the IPC and miss rate results.

2 Related Work

An optimal replacement strategy has been proposed by Belady [2] which replaces a block that is referenced the farthest in the future. LRU (Least Recently Used) has been shown to be a good practical alternative. Various cache replacement strategies have been proposed, but none of them are consistently superior to others.

[1] Using the methodology described in Section 4.

[2] Because the simulator does not update cache replacement state when a writeback access is a hit, the trace file we generated does not contain writebacks that are hits.

PLRU is a simplified implementation of LRU based on heuristics to select the block to be evicted upon a miss. [1] compares the performance of PLRU (Pseudo LRU) versus the performance of LRU and Round-Robin. [11] proposed $LRU - K$, an extension of LRU based on the observation of the past K references. [9] proposed LRFU (Last Recently Frequently Used) as a parametric replacement strategy that attempts to take advantages of both LRU and LFU (Last Frequently Used) policy. LRFU weigh blocks by using a weight monotonic function. This function depends from a single parameter that can assume values into a continuous limited interval. Each value of this parameters corresponds to one LRFU policy with time complexity to select the block to evict ranging from LRU (constant) to LFU (logarithm of the number of blocks).

A different approach is to implement an adaptive policy able to select the best policy for a given benchmark during benchmark execution. [3] proposed a replacement strategy based on virtual caches. Virtual cache areas of main memory are used to maintain the state of many different well known policies during program execution, i.e. RAND, FIFO, LIFO, LRU, MRU, LFU, and MFU. Each virtual cache simulates the operation of a single baseline policy. A master policy observes the miss rate on the actual request stream and switches dynamically to the policy that seems to be the most suitable at a given time. [12] introduced three adaptive strategies as extensions of LRU. The aim of these algorithms is to prevent cache thrashing for workloads with a large memory footprint. The first strategy is called LIP (LRU Insertion Policy). LIP is an insertion policy that places all incoming lines in the LRU position, and moves them to the MRU position only if they are referenced while they are still in the LRU position. The second algorithm is BIP (Bimodal Insertion Policy). In BIP, some of the incoming lines are placed directly in the MRU position. BIP improve LIP in the mean that it adapts to changes in the working set during the program execution. Since both LIP and BIP gain in performance against LRU on some benchmark but perform poorly on others, a dynamic policy, called DIP (Dynamic Insertion Policy), was proposed to choose dynamically between the traditional LRU policy and BIP depending on which policy has fewer misses.

All previous approaches, including Belady's algorithm, consider miss penalties to be equal in cost. Replacement strategies called CSOPTs (Cost-Sensitive OPTimal replacement algorithms) [8], assume that cache misses are not equal in cost. CSOPTs are off-line, so they are proven to approach a better lower bound, in terms of cache misses, than the Belady algorithm. Practical implementations extending LRU, and approximating CSOPT, have also been proposed.

3 The SCORE Replacement Policy

SCORE assigns each cache line a score which reflects access behavior in the past. Lines with low scores are chosen as victims for replacement. There are two different actions: in the front-end, the scores are computed dynamically, and in the back-end, the scores are used to replace lines.

In our system, the lowest score is set to zero. If the number of bits is n, the highest score will be $SCORE_{MAX} = 2^n - 1$. The more number of score bits will allow higher resolution, hence increase the precision of the SCORE policy. For LRU or LIP and a cache associativity of 16, each line within a set has a unique score from 0 to 15. The number of score bits for these replacement policies is 4. Our SCORE policy, meanwhile, allows the number of bits to be determined at design time, offering higher precision.

3.1 Setting the initial scores

After a new line is brought from the main memory, it is assigned an *initial score*. Lines which will be accessed again in the near future should have high initial score to keep them in the cache long enough. This happens in the applications with high locality. Meanwhile, lines which will not be accessed again or will be accessed again only after a long time have low initial scores, so that they will be evicted and replaced by new lines. This applies to applications with low locality, such as pointer-intensive applications. The specific value of initial score should vary not only according to different applications, but also different phases of execution (e,g., different loops). In the best case, for each execution phase, one should know which cache lines will be reused to assign a high initial score, and which ones will not be accessed again to assign a low initial score. Interestingly, the best initial scores for an application are not always the highest or the lowest score (like in LRU or LIP), but fall somewhere in the middle. This is because a typical application has both lines with high and low locality. By analyzing the miss rates with different initial scores, one can achieve the best average initial scores.

Our results show that, for a given application, the best initial score can take any value from 0 to $SCORE_{MAX}$. This means that, the initial scores should be application-specific, and even phase-specific. Hence, we investigate dynamic initial scores. The dynamic initial scores are determined by monitoring the misses over a fixed number of cache accesses (intervals) and adjusting the score based on that number. We set a step, which is added to or subtracted from the current initial score according to the changing direction. During the current interval, if the number of misses is smaller than that of the previous interval, we keep the direction of change and move one more step forward. Otherwise, the direction is changed,

and the initial score is changed by an amount equal to the step.

3.2 Changing the scores

Another property of the score-based policy is to change of the scores based on the access to the cache. We study the velocity of change, which is the speed of changing scores when a line is hit or missed. If a line is hit, its velocity score is increased by a number equal to the *increase velocity*. We also define *decrease velocity*, which is the speed of decreasing the line score due to the replacement of lines or the hit of another line in the same set. When a new line is brought from the memory, or is hit, scores of all other lines are decreased. To avoid the scores from going too low, the increase velocity is higher than the decrease velocity.

Let us consider LRU, MRU and LIP in terms of velocity. Increase velocity, for the hit line, is equal to the difference between the highest score and current score of that line. Decrease velocity, which is equal to 1, is applied to all the lines with scores higher than the score of the hit line or the evicted line. Our policy, meanwhile, have fixed increase scores, and score increase does not always make the score of a line reach $SCORE_{MAX}$. This is because not all the lines after being accessed will be accessed again in the future.

Initial scores and change velocities are the two important properties for the score-based policy. Another property, which is applied to all the lines of the same set, is the summation of all scores of all lines of the set. Initial scores and changing velocity chosen so that the total score of a line is not too high or too low.

3.3 Choosing the victim lines

Decision on which line to evict is done by comparing the scores of the lines in the same set. Lines with low scores have not been hit in the past, and are good candidates to be evicted. Normally, when choosing one line, the line with the lowest score will be a good candidate. However, it can be observed that lines with low scores have not been hit for a while, and they are all good candidates to be replaced. Hence, in SCORE, we set a *threshold score*. The random policy is applied to the lines under threshold, as these lines contribute equally in the hit rate, and choosing the line with the lowest score is not always the best choice.

In the case when no lines have scores lower than that threshold, the line with the lowest score is chosen. This is similar to the LRU policy.

4 The methodology

The following SPEC CPU 2006 [6] benchmarks were compiled under Linux operating system using GNU compilers with optimization flag -O3 for x86_64 architecture: 401.bzip2, 403.gcc, 429.mcf,434.zeusmp, 435.gromacs, 436.cactusADM, 437.leslie3d, 447.dealII, 450.soplex, 456.hmmer, 473.astar. The traces were collected using Pin [10]. Traces for 500M instructions and traces for 5B instructions are collected after the first 40B instructions are skipped. The simulation framework is based on CMP$im [7], that is a multi-core cache simulator based on Pin [10]. For the purpose of this work CMP$im models a simple out-of-order core and a memory model consisting of a 3-level cache hierarchy as described at `http://www.jilp.org/jwac-1/framework.html`.

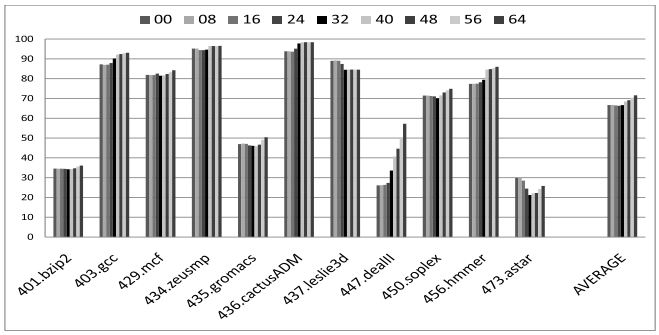

Figure 2. Impact of Initial Scores on Miss Rates.

5 Results and Analysis

We present the impact of initial scores, dynamic initial scores and threshold on miss rates on 500M instruction traces. Finally, we show miss rates and speed up over well-known policies using 5B instruction traces. Our results are presented for 6-bit scores ($SCORE_{MAX} = 64$), increase velocity of 40 and decrease velocity of 1.

Figure 2 shows the impact of initial score on the miss rates. One can conclude that the best initial score depends on the application. Hence, a dynamic initial score, which is adaptive to the application, is needed.

We studied adjustment intervals of 10K and 100K accesses, with three different steps 2, 4 and 8. Results showed that, the interval of 100K accesses has lower miss rate.

In Figure 3, with the dynamic initial step, we present the impact of different thresholds on the miss rates. The threshold of 0 creates the behavior similar to that of LRU, while the threshold of 64 creates the random selection behavior. The average miss rates show that the best threshold, which produces the lowest miss rate, falls in the range from 16 to 32. If we look at the benchmarks individually, there are two general trends. The first trend is the increase of miss rates as the threshold increases. This happens in the benchmarks which have good locality, as some lines which will be accessed again in the short future, is replaced as their scores fall under the threshold. The benchmark astar sees a sudden change in the miss

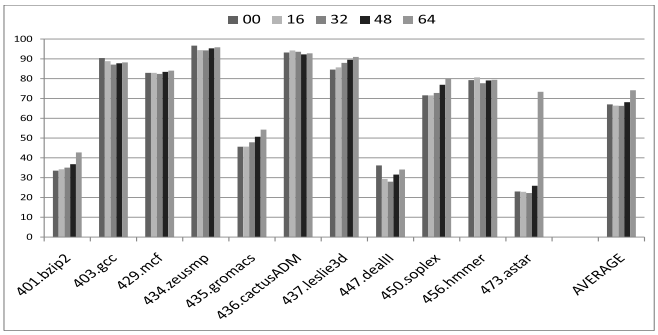

Figure 3. Impact of Thresholds on Miss Rates.

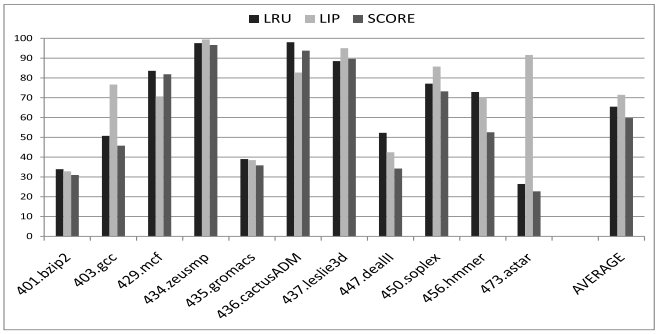

Figure 4. Miss Rates of Different Policies.

rate in the full random selection policy. The other trend is similar to the trend of the average, with the lowest miss rates in the middle.

The above results are presented for the 500M instruction traces. In the rest of this section, we use 5B instruction traces.

From the above results, we choose the threshold of 24, and compared our SCORE policy with LRU and LIP in terms of miss rates and IPCs. Figure 4 compares the miss rates of these policies. In the set of 11 benchmarks, SCORE has better miss rates in 10 benchmarks, and only loses slightly in leslie3d. It enjoys big reduction for two benchmarks dealII (18.03%) and hmmer (20.36%). Compared to LIP, SCORE has 9 benchmarks which have smaller miss rates. Significant benchmarks include gcc (30.85%), dealII (8.19%), soplex (12.51%), hmmer (17.81%), and astar (68.89%). However, on two benchmarks, mcf (11.17%) and cactusADM (11.08%), LIP wins. This means that, there is still room for improvement in our policy. We will look into the dynamic velocity and threshold in the future work. Figure 5 compares the absolute IPCs for these policies. IPC results show consistency with miss rates.

In terms of speedup over LRU, SCORE has the standard deviation of 0.10, and LIP has the standard deviation of 0.2. This confirms the fact that SCORE is more stable than LIP. It can also be confirmed from Figure 5, where LIP has some benchmarks with high improvement, but also has other benchmarks lose big over LRU and

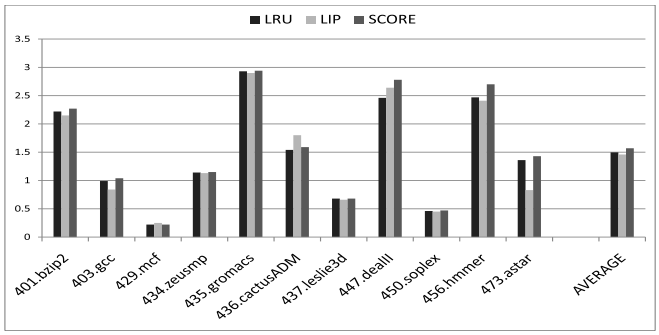

Figure 5. IPCs of Different Replacement Policies.

SCORE.

Finally, the average IPC improvement of SCORE over LRU is 4.9%, and over LIP is 7.4%.

References

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, New York, NY, USA, 2004. ACM.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, 1966.

[3] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari. Adaptive caching by refetching. In *NIPS*, pages 1465–1472, 2002.

[4] D. Grund and J. Reineke. Estimating the performance of cache replacement policies. In *MEMOCODE '08: Proceedings of the 6th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 101–111, June 2008.

[5] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. *SIGMETRICS Perform. Eval. Rev.*, 34(1):228–239, 2006.

[6] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[7] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps. Technical report, 2006.

[8] J. Jeong and M. Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Trans. Comput.*, 55(4):353–365, 2006.

[9] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, 2001.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[11] E. J. O'neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering, 1993.

[12] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 381–391, New York, NY, USA, 2007. ACM.