Situation-based Cache Replacement Policy

Meera Suthar (ms12418) and Harsh Patel (hp2178)

New York University

December 1, 2021

Abstract

The shared last-level cache plays a significant part in further developing application execution and diminishing off-chip memory data. To utilize LLCs all the more proficiently, ongoing research and past literature have shown that changing the re-reference expectation on cache additions and cache hits can altogether further develop cache performance. The practical cache replacement algorithm tries to imitate optimal substitution by foreseeing the re-reference interval length. The ordinarily utilized LRU substitution strategy consistently predicts a close to prompt re-reference span on cache hits and misses. We propose a Situation-based Cache Replacement Policy (SBA) that dynamically chooses one of the policies among Least Recently Used, Static RRIP ([1]), and LFU based on the situation. Based on the experiments on SPEC CPU2006 programs, Situation-based Cache Replacement Policy (SBA) outperforms at least one of SRRIP or DRRIP ([1]) for a good number of program traces.

Keywords
Cache Replacement Policy, Low-Level Cache, Shared Cache, Situation-based Cache Replacement Policy

1 Introduction

An ideal replacement strategy settles on its replacement decisions by utilizing information on the reuse of each cache block and replaces the block that will be re-referred to farthest later on. Typically, on a miss, replacement policies build a prediction on a cache block that will not be used in near future. Such block is evicted and a new block is inserted in the cache.

When multiple similar cores share the same shared cache, the replacement policy should be such that it only targets the cache line having the data of the same core. If a cache contains various blocks of different sizes, then it is beneficial to evict one big block rather than replacing multiple smaller blocks. In addition to that, the inclusive and exclusive properties of a cache may also affect the hit ratio. The main goal of any shared or non-shared cache should be to reduce the overall execution time. To reduce the gap between main memory and CPU, a hierarchy of caches is introduced. If a cache replacement policy evicts the set of blocks which were supposed to be used by the program in the near future, then it will cost the program a significant amount of time.

Most of the cache replacement policies consider temporal data while selecting the victim when a cache miss occurs. The LRU chain depicts the recency of cache blocks documented in the commonly used Least Recently Utilized (LRU) replacement policy, with the MRU position reflecting a cache block that was last used and the LRU position representing a cache block that was least recently used. The LRU chain's outline has been updated by recent ideas on cache insertion policies and hit promotion strategies.

While the LRU policy provides good performance for high-locality workloads, it has pathological behavior for memory-intensive workloads with an operating set bigger than the available cache capacity. Also, it performs poorly when we have the program has distant re-reference intervals. This situation leads to cache thrashing. The Least Frequently Used algorithm selects a block having the least number of cache re-references. It does not utilize temporal data while evicting the block from a cache line. So, LFU is a scan-resistant algorithm, but over time, it also introduces stale blocks issues.

There are various proposals to improve LRU performance; however, several of these come with significant storage overhead, significant modifications to the current style, and poor performance for LRU-friendly workloads. Each new structure and change to an existing design necessitates design, verification, and testing efforts. The goal of this study is to develop a cache replacement policy that works well for both LRU-friendly and LRU-averse workloads while needing minimal hardware changes and overhead.

We present our study of a hybrid cache replacement policy that works for varied workloads and access patterns aiming to develop a robust policy that can dynamically choose the optimal cache block for replacement. We present our remaining study as follows: Section 2 gives insights into the previous research related to this field of study. Section 3 introduces a Situation-based Cache Replacement Policy (SBA). The information about the experiments and benchmarks used, results, and analysis of our research is given in Section 4 and 5, which is followed by a conclusion in Section 6.

2 Literature Review

Over the last few years, much research had been carried out regarding a novel and optimal Cache Replacement Policy. Significant part of the research was focussed on modifying the most common replacement, which is LRU and Belady's optimal replacement policy. We will present the past works below which helped us directly or indirectly to achieve our results in the experiments.

Authors in paper [2] discuss the utilization of unconstrained profound RNN to get the intuitions of cache design and afterward constructing a minimal expense basic cache predictor which can be executed in hardware and trained progressively. One of the downside of this approach is that such deep learning models are impractical for hardware cache replacement. Because deep learning models are impractically large and sluggish, they are unsuitable for use in hardware predictors. So, the authors have suggested to use the model offline and then use the insights from that to design the hardware cache. In paper [3], authors proposed an algorithm called Frequency-Based LRU which takes the recent access information, partition and the frequency information into consideration along with that the authors suggest to partition the shared cache into N independent parts corresponding to each of the cores in order to avoid cache conflicts. If the cache is partitioned directly into N parts based on the number of cores then, this approach is not useful for applications which requires working set larger than the small subpart of the shared cache. Because for such application there would be cache miss most of the time compared to cache hits.

As per the authors of paper [4], the reciprocal of a cache block's reuse distance and its projected hit count have a significant relationship. Expected Hit Count (EHC) policy for replacement of cache blocks in last-level caches is an effective, low-cost replacement strategy that associates a predicted hit count with a block and aims to evict the block that is expected to get fewer hits in the future, based on this observation. While in paper [5] authors discusses an improvement of Hawkeye replacement algorithm in which the algorithm distinguishes prefetches from demand fetches, allowing redundant prefetches to be recognized and suitably cached. Even though the results show that the modified Hawkeye policy outperforms the standard replacement policies, we believe that this change is practically too small. This could be due to the fact that we still don't have an optimal replacement policy in presence of prefetches.

In papers [6] and [7], authors proposes a modification of LRU replacement policy and Belady's optimal replacement policy respectively. Former uses a temporal locality-aware modified LRU replacement technique. Non-temporal lines will be selected as victims over temporal lines by the replacement policy. If all non-MRU lines are temporal, the victim is chosen based on the LRU replacement policy instead of the MRU cache line. (i.e., a probability indicating whether a cache block will be reused) by combining two techniques: profiling and dynamic detection. Later one proposes an imitation learning strategy to automatically learn cache access patterns based on prior accesses that correctly approximates Belady's algorithm even on diverse and complicated access patterns in order to compute the best eviction decision given future cache accesses.

Most of the cache replacement policies focus on improving a single cache, but very few papers focus on improving hit ratio over a hierarchy of caches. Because of the independent nature of each cache in the hierarchy, the overall performance of hierarchy of caches will be affected. A global cache replacement policy has proposed in paper [8].

Two novel hashing functions, prime modulo and prime displacement, are described in paper [9], which are resistant to pathological behavior of cache hashing functions and can prevent worst-case conflict behavior in the L2 cache. Prime displacement hashing adds an offset equal to a prime number multiplied by the tag bits to the index bits of an address to produce a new cache index, whereas prime modulo hashing employs a prime number of sets in the cache. These hashing approaches can be implemented on fast hardware that use a set of narrow add operations rather than actual integer division and multiplication. For the L2 cache, this solution has very little fragmentation. This paper [10], which gives a detailed examination of the predictability of the LRU, FIFO, MRU, and PLRU policies in terms of three metrics, evict, fill, and mls, that capture features of cache-state prediction, was also extremely informative and useful for our experiments.

3 Situation-based Cache Replacement Policy (SBA)

In any cache replacement policy, we have to take care of two things: hit ratio and access latency. If we want to improve the hit ratio of a policy, we need more details (bits/flags) to evict any cache line. But that may result in more access latency. LLC caches are shared caches and are used by multiple cores on a single machine. Our goal is to minimize the number of cache misses. If we have more cache hits and fewer cache misses, then the number of queries made to fetch the data from the main memory will be reduced. We propose a hybrid situation-based cache replacement policy that utilizes the pros and cons of various existing cache replacement policies and makes a decision to switch between policies. The proposed cache replacement policy uses LRU, SRRIP ([1]), and LFU cache replacement policies and makes a transition from one policy to another based on various conditions shown in Figure 1.

The algorithm starts with the LRU policy. LRU is an efficient policy when it comes to evicting cache lines based on time. The cache line which is Least recently used will be evicted among all. But the problem with the LRU-based replacement policy is that it gives us the worst performance when we have distant re-reference intervals. Say, the cache size is k, and the re-reference distance is d. If d is lesser than k, then

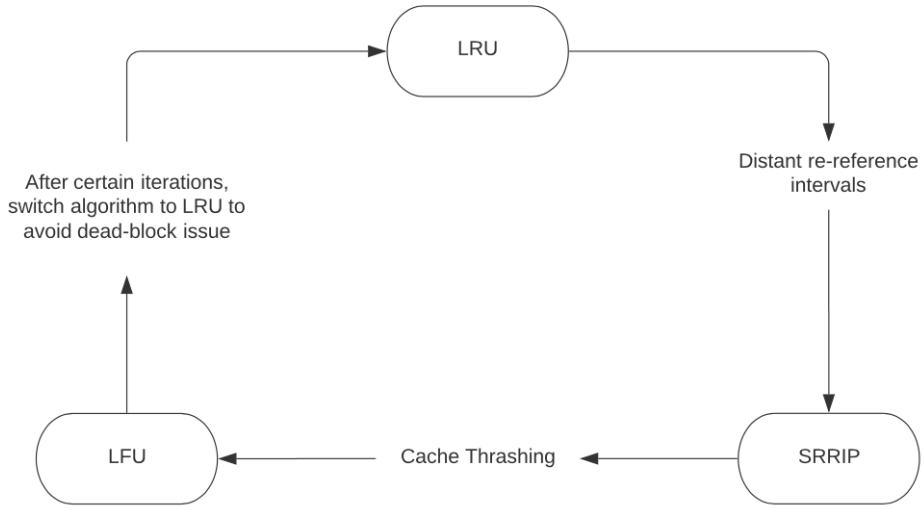


Figure 1: Situation-based Cache Replacement Policy (SBA)

it is okay to use LRU. But say for most of the cache lines, if the distance d is greater than k, then LRU will evict the frames which were going to be used afterward. To avoid this issue, the algorithm makes a transition from LRU to Static Re-reference Interval Prediction (SRRIP) policy. The decision of switching from LRU to SRRIP will be taken if a certain number of continuous cache misses has occurred. We tried various threshold values (the minimum number of continuous misses should occur) for switching the policy from LRU to SRRIP and got good results for the threshold value 25.

SRRIP is a scan-resistant policy that gives good results when we have re-reference intervals of various lengths. It assigns a cache line a number that ranges from 0 to maxRRPV (maximum re-reference prediction value - which is mostly in terms of $2^m - 1$); where 0 represents highest priority and maxRRPV represents the lowest priority. The cache line having the value maxRRPV is selected as a victim. If a cache hit occurs, the RRPV value for that cache line will become 0. If a miss occurs, then the RRPV value for that particular cache line will become maxRRPV-1. During a cache miss, if none of the cache lines has a value maxRRPV, then the RRPV value of all cache lines will be incremented by one and again search will be continued with updated RRPV values until the target cache line is found.

But again, SRRIP is not a good choice when all the cache lines have re-reference intervals larger than the cache size. In that case, cache thrashing will occur in the case of SRRIP. To avoid this issue for a longer time, our algorithm will make a transition from SRRIP to LFU when such scenario occurs. The algorithm identifies cache thrashing with a higher number of continuous cache misses. The threshold value which we kept here is 50. We can also set a threshold for non-continuous cache miss count up to a certain number to switch the algorithm.

By making a switch to LFU, we are avoiding the age and timing factor while evicting a block from the cache. LFU starts evicting the blocks using hit frequencies rather than temporal information and that will again increase the performance. But the issue with LFU algorithm is that - it is not a good thrash resistant algorithm. Also, LFU comes with dead block issue. If we have more blocks which was used most frequently in the past and are now no longer in use, the algorithm may never evict such blocks though they are not currently being used. To avoid this situation, after a certain period (50 iterations), our proposed algorithm will make a switch to again LRU policy. Looping various cache replacement policies will avoid cons of any particular cache replacement policy and give us the better results than any particular single policy.

4 Experimental Setup

We utilized ChampSim simulator to perform our experiments. ChampSim is a trace-based simulator which measures IPC, CPU instructions and cycles, Simulation time, Cache-wise number of misses, hits and average miss latency. We utilized the measurements related to LLC cache and compared the results of various algorithms. The comparison was done using the cache miss percentage reduction formula which is shown below:

$$\text{Cache miss \% reduction} = \left(\frac{M_{base} - M_{current}}{M_{base}}\right) * 100$$

Here, M_{base} specifies number of misses in base algorithm and $M_{current}$ specifies number of misses in the current algorithm for which we try to evaluate the results. We used DPC-3 traces provided by Professor Daniel Jimenez at Texas A&M University to perform

experiments and compare results. DPC-3 traces contains SPEC CPU2006 programs which has a good variety of patterns. We utilized a single-core processor with bimodal branch predictor, and no L1/L2 data prefetchers. LRU algorithm was used as a base algorithm to perform the experiments.

5 Experiments and Analysis

We compared Situation-based Cache Replacement Policy (SBA) with Static Re-Reference Interval Prediction (SRRIP) and Dynamic Re-Reference Interval Prediction (DRRIP). LRU is the base algorithm that uses temporal information while making cache eviction decisions. SRRIP is a scan-resistance algorithm that not only uses temporal information but also utilizes re-reference interval information to make a decision. DRRIP is more robust again scan-resistance and thrash-resistance. The measure used to compare two or more replacement policies is Cache Miss % Reduction. In SBA, we tried the Clock algorithm (an approximation of LRU), Not Recently Used policy (looks at the usage of last clock interval rather than looking over a short period of time) and SIP (decreases the block priority by 1 rather than updating it to priority 0 on cache hit) replacement policies as an alternative of LRU but got the best results by keeping LRU. Also, we also developed an algorithm that only utilized two replacement policies to switch (SRRIP and LFU) but did not notice an improvement over SBA. Also, we tried various threshold values for continuous misses for LRU and SRRIP and time counter for LFU but got the best results for continuous miss count 25 for LRU, continuous miss count 50 for SRRIP, and time period 50 for LFU.

We only updated the replacement policy for LLC shared cache and compared the results. We utilized seven of twenty SPEC CPU2006 traces (which showcased the significant change in numbers) to compare the results of selected algorithms. The SPEC CPU2006 traces used were - mcf, cactuBSSN, lbm, omnetpp, cam4, pop2 and fotonik3d (Figure 2).

Out of 20 SPEC CPU2006 program traces, the results for 7 program traces are displayed. The graphs (Figure 2) show the cache miss % reductions for 7 different traces. SBA beats SRRIP for programs lbm, cam4 and fotonik3d. SBA also gives better results than DRRIP in the case of cactuBSSN and pop2. For mcf and omnetpp, SBA gives the same results as SRRIP. We can conclude here that SBA gives results somewhere between SRRIP and DRRIP.

6 Conclusion and Future Work

We introduced a Situation-based Cache Replacement Policy (SBA) that dynamically chooses one of the policies among Least Recently Used, Static RRIP ([1]), and LFU based on the situation. We assessed our algorithm on mcf, cactuBSSN, lbm, omnetpp, cam4, pop2, and fotonik3d programs from SPEC CPU2006. Based on our experiments, we observe that our dynamic replacement policy outperforms at least one of SRRIP or DRRIP ([1]) on most of the programs. This strategy can be extended for various algorithms such as NRU, SIP, and other advanced algorithms. Also, we can utilize time series machine learning models such as RNN, LSTM to learn temporal patterns of the program. This subsequently helps the replacement policy to choose the appropriate victim and that results into a better hit ratio. Also, instead of switching between three policies, we can extend the algorithm to switch between n policies, but this also will come with some limitations.

References

[1] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel S. Emer. High performance cache replacement using re-reference interval prediction (rrip). *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[2] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 413–425, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Fang Juan and Li Chengyan. An improved multi-core shared cache replacement algorithm. In *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*, pages 13–17, 2012.

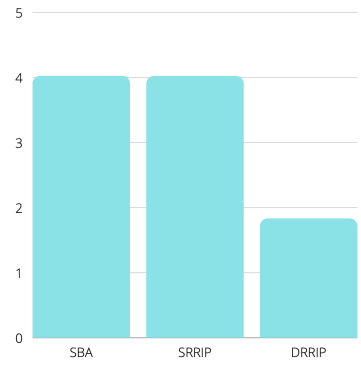
[4] Armin Vakil-Ghahani, Sara Mahdizadeh-Shahri, Mohammad-Reza Lotfi-Namin, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Cache replacement policy based on expected hit count. *IEEE Computer Architecture Letters*, 17:64–67, 2018.

[5] Akanksha Jain and Calvin Lin. Hawkeye: Leveraging belady's algorithm for improved cache replacement. 2017.

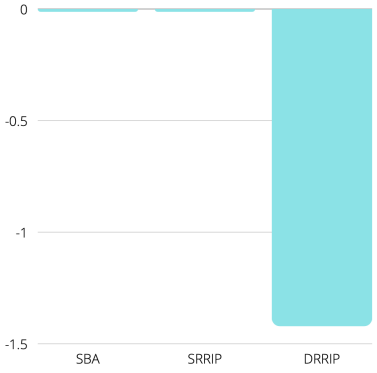
[6] Wayne A. Wong and Jean-Loup Baer. Modified lru policies for improving second-level cache behavior. *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, pages 49–60, 2000.

[7] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. *ArXiv*, abs/2006.16239, 2020.

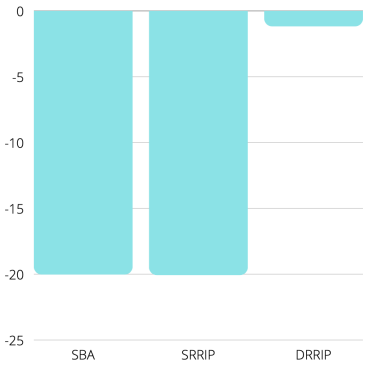
[8] Mohamed Zahran. Cache replacement policy revisited. 01 2007.



(a) mcf

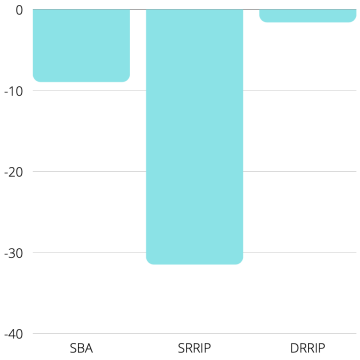


(b) cactuBSSN

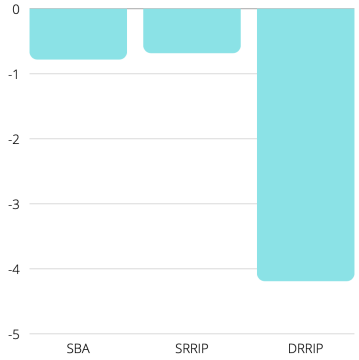


(c) lbm

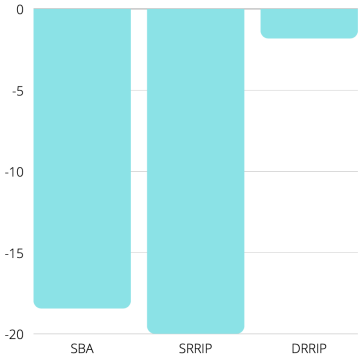
(d) omnetpp



(e) cam4



(f) pop2



(g) fotonik3d

Figure 2: Cache miss % reduction for SBA vs SRRIP vs DRRIP (LRU as a base algorithm)

[9] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using prime numbers for cache indexing to eliminate conflict misses. *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 288–299, 2004.

[10] Jan Reineke, Daniel Grund, C. Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, 2007.