Another naïve implementation of SHiP++ based Last-Level Cache Replacement Algorithm

Sachin Shreenivas Puranik,
Department of ECE,
Texas A&M University, College Station, TX 77840
Email: pura247@tamu.edu

Abstract:

Cache replacement policies serve the purpose of predicting memory access patterns to make the efficient use of cache to reduce latency and the miss rates. The shared last-level caches in CMPs play an important role in improving application performance and reducing off-chip memory bandwidth requirements. Early cache replacement policies like LRU simply predict the hit and miss blocks will be re-referenced immediately afterward. However, this pattern of replacement does not apply to LLC as the accesses are filtered by the higher level of memory hierarchy. To properly adapt to these applications, researchers proposed a novel cache replacement policy, Signature-based Hit Predictor (SHiP) to learn the re-reference behavior of cache lines belonging to each signature thereby providing thrash, scan, and streaming resistance. While we find that SHiP offers substantial improvements over the baseline LRU replacement and state-of-the-art replacement policy proposals, we identify few areas of improvement. So, in this paper, we present the design and our implementation of SHiP++ cache replacement policy, proposed by the same authors that proposed SHiP. We also evaluate the SHiP++ on the provided CMPSim infrastructure in comparison to other policies like LRU. From our implementation, we can see that SHiP++ improves performance by **6.53** percent over the LRU.

Introduction and Related Work:

Cache is an indispensable intermediate level storage in the memory hierarchy of modern computer architecture. It fills the great speed gap between high-performance processors and the main memory. Cache size is always limited because it's too expensive, thus it can only take a small, frequently-used portion from main memory to feed the processor in a much quicker manner. Therefore, there must be sometimes that the processor wants to access the portion of memory that is not currently available in the cache, namely, this is a cache miss. The widespread use of chip multiprocessors with shared last level caches (LLCs) and the widening gap between processor and memory speeds increase the importance of a high performing LLC. Recent studies, however, have shown that the commonly-used LRU replacement policy still leaves significant room for performance improvement. As a result, a large body of research work has focused on improving LLC replacement. This paper focuses on improving cache performance by addressing the limitations of prior replacement policy proposals.

The commonly-used LRU replacement policy predicts that all cache lines inserted into the cache will have a near-immediate re-reference interval. However, such a prediction causes poor performance for applications that have frequent bursts of references to non-temporal data like scans. [1] and [2] have illustrated that the worst cases of LRU comes during the following memory access patterns:

- Thrashing
- Streaming
- Mixed

[1] has also illustrated that for both thrashing and mixed access patterns, there is room to improve LRU. The RRIP [1] policy is designed to protect the cache from these patterns. Figure 1 shows the state diagram for RRIP, where each line is equipped with a 2-bit counter to track the Re-Reference Interval Prediction Value (RRPV).

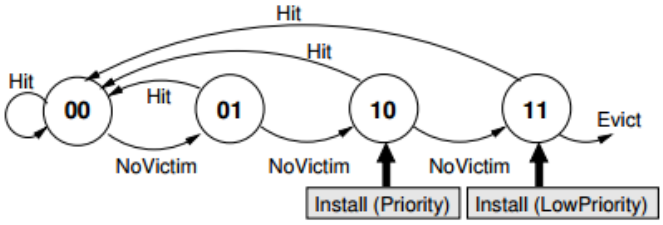

Fig.1 Re-Reference Interval Prediction

On a hit to the line, the RRPV is reset to 0. On a miss, the victim is found by searching from way 0 and finding the first line in the set with RRPV of 3. If no such line is found, the RRPV of all lines in the set is incremented and the search is repeated. RRIP can either insert the line with a high-priority position (RRPV=2) or a low-priority position (RRPV=3). The choice of insertion position can be determined either statically for all references (SRRIP) or dynamically (DRRIP) at runtime.

The performance of the SRRIP component policy is dependent on two important factors. First, SRRIP relies on the active working set to be re-referenced at least once. Second, SRRIP performance is constrained by the scan length. For short scan lengths, SRRIP performs well. However, when the scan length exceeds the SRRIP threshold or when the active working set has not been re-referenced before the scan, SRRIP behaves similarly to LRU replacement. SHiP focuses on designing a low overhead replacement policy that can improve the performance of mixed access patterns. SHiP++ further tries to enhance the drawbacks present in the SHiP. Our experimental results show that SHiP++ performance is significantly better than the commonly-used LRU policy. This results in improving performance by 6.5% over LRU, and by 1.8% over SHiP as seen from geometric mean IPC in Figure 2.

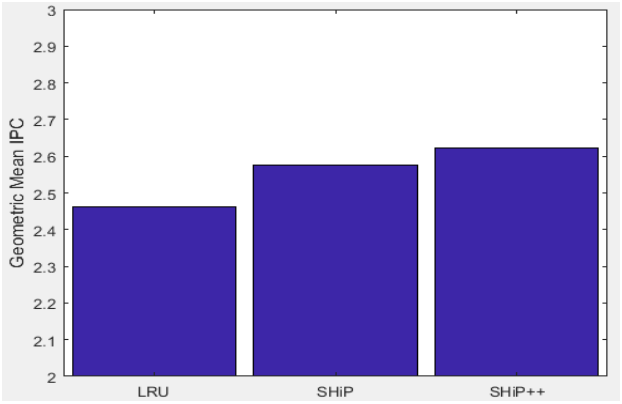

Fig.2 Geometric Mean IPC values of LRU, SHiP, and SHiP++

Signature-based Hit Predictor:

i. Reason:

When choosing the replacement policies, we first narrowed down several high-quality research papers, which are RRIP, SDBP, SHiP, RRP, Perceptron Learning for Reuse Prediction, SHiP++ and Multiperspective Reuse Prediction among which the RRIP is the earliest; Multiperspective Reuse gives great accuracy but difficult to implement given the tight schedule. After a thorough contemplation, we chose SHiP++ for the following reasons:

- Re-Reference Interval Prediction is too easy and speed-up over LRU is not very high and would not help me win the competition.

- We tried Sampling Dead Block Predictor thinking that we would extend it to Perceptron or Multi-perspective but after implementing SDBP paper we couldn't get much performance out of SDBP. It was only 1.8% due to several underlying bugs in our implementation.
- Multi-perspective Reuse has the best performance of all the policies but its ultra-hard to implement in the given timeline.
- RRP and RWP implementations are too ambiguous and may not give a good performance.
- SHiP++ is the latest policy by the group that implemented SHiP and RRIP, gives a good speed-up and enjoys a low latency while occupying a reasonable space.
- SHiP++ is easy to implement both in a program and on the hardware, which means it's highly applicable.

ii. Design:

Now it's time to introduce SHiP++ [3] design. In this project, we are implementing the enhancements proposed by the SHiP++ [3] paper on top of the SHiP[2] algorithm.

The goal of signature-based cache replacement is to predict whether the insertions by a given signature will receive future cache hits. The intuition is that if cache insertions by a given signature are re-referenced, then future cache insertions by the same signature will again be re-referenced. Conversely, if cache insertions by a given signature do not receive subsequent hits, then future insertions by the same signature will again not receive any subsequent hits.

To learn the re-reference pattern of a signature, SHiP requires two additional fields to be stored with each cache line: the signature (We have chosen Program Counter (PC) for this project) itself and a single bit to track the outcome of the cache insertion. The outcome bit (initially set to zero) is set to one only if the cache line is re-referenced. SHiP also uses Signature History Counter Table (SHCT) of saturating counters to learn the re-reference behaviour of a signature. When a cache line receives a hit, SHiP increments the SHCT entry indexed by the signature stored with the cache line. When a line is evicted from the cache but has not been re-referenced since insertion, SHiP decrements the SHCT entry indexed by the PC signature stored with the evicted cache line.

As for the replacement policy, SHiP can be used in conjunction with any ordered replacement policy. The primary goal of SHiP is to predict the re-reference interval of an incoming cache line. We evaluate SHiP using the SRRIP with Frequency priority (FP) replacement policy. We use SRRIP because it requires less hardware than LRU and in fact outperforms LRU [1]. In the absence of any external information, SRRIP conservatively predicts that all cache insertions have an intermediate re-reference interval. If the newly-inserted cache line is referenced quickly, SRRIP updates the re-reference prediction to near-immediate; otherwise, SRRIP downgrades the re-reference prediction to distant. SHiP on the other hand explicitly and dynamically predicts the re-reference interval based on the SHCT. SHiP makes no changes to the SRRIP victim selection and hit update policies. On a cache miss, SHiP consults the SHCT with the PC signature to predict the re-reference interval of the incoming cache line.

While SHiP performs well by learning the re-reference characteristic of PCs, SHiP++ enhance SHiP re-reference predictions at cache insertion and cache hits. Furthermore, SHiP++ also enhances the training for the Signature History Counter Table (SHCT) on cache hits. The first enhancement comes from a fact that, SHiP always installs with state 2 or 3, but, some workloads benefit from keeping re-use lines longer. Therefore, SHiP++ uses SHCT to confidently install lines at state 0, when SHCT is saturated at the highest count value. The second enhancement comes from the observation that, SHCT Learns on all hits and evictions. This can be problematic because a small number of high-access-frequency lines saturate SHCT entries; solution SHiP++ proposes is to learn from only first-hit and evictions. Third improvement suggested by SHiP++ is to install writebacks at state 3. This comes from the observation that writebacks are not in critical path and can be bypassed. Final two enhancements are based on the prefetching model: First one comes from the fact that SHiP algorithm doesn't differentiate between prefetch and non-prefetch accesses. But the intuition is that demand may have re-use, but prefetched lines may not have re-use. Hence, SHiP++ proposes to learn separately in different halves of SHCT for prefetch and non-prefetch accesses. Final enhancement proposed by SHiP++ comes from the observation that, prefetches are staying in caches for a long time. First-access to the prefetched line is a demand access. Original SHiP promotes and keeps accurate prefetches past usefulness. Hence, SHiP++ proposes to ignore state-update for first access to the prefetched line and update for subsequent accesses.

The algorithm below summarizes the cache insertion, promotion and replacement policies of the combined SRRIP, SHiP and SHiP++ schemes that have been implemented in our project.

iii. Algorithm:

Algorithm: SHiP++ Policy using SHiP and SRRIP.

```
function INITIALIZE
        numSets = number of sets in the cache
        assoc = cache associativity
        init replInfo[numSets][assoc]
        for sIndex = 0 → (numSets-1) do
                for wIndex = 0 → assoc-1 do
                        repl[sIndex][wIndex].SHiP_PC =0
                        repl[sIndex][wIndex].outcome =0
                        repl[sIndex][wIndex].is_pf =0
                        repl[sIndex][wIndex].interval = FAR_INTERVAL
                end for
        end for
        set SHCT ← 0
end function

function GetMyVictim(sIndex)
        victim_number = assoc – 1
        victim_found = false
          while victim_not_found do
                for wIndex = 0 → assoc do
                        if repl[sIndex][wIndex].interval = DISTANT_INTERVAL then
                                victim_found = true;
```

```
                    return victim_number;
                end if
                if vitim_found = false then
                    for wIndex = 0 → assoc do
                        repl[sIndex][wIndex].interval++ if repl[sIndex][wIndex].interval < THRESHOLD
                    end for
                end if
            end for
        end while
end function

function UpdatePolicy(PC, sIndex, updateWayID, cacheHit, accessType)
    signature ← (PC << 5) + ((accessType == ACCESS_PREFETCH) << 4) ..) & MASK;
    if cacheHit and (repl[setIndex][updateWayID].outcome == false) then
        if repl[setIndex][updateWayID].is_pf = true then
            repl[setIndex][updateWayID].is_pf = false
        else
            Increment SHCT[signature] if counter  < THRESHOLD
            repl[sIndex][wIndex].interval-- if  repl[sIndex][wIndex].interval > 0
        end if
    else
        if evicted_cache_line.outcome != true then
            Decrement SHCT[signature] if counter  > 0
        end if
        repl[setIndex][updateWayID].outcome = false;
        repl[setIndex][updateWayID].SHiP_PC = signature;
        repl[setIndex][updateWayID].is_pf = (accessType == ACCESS_PREFETCH);

        if SHCT[signature] = 0 or access = writeback then
            Predict distant re-reference
        else if SHCT[signature] = HIGHEST_COUNT
            Predict IMMEDIATE re-reference
        else
            Predict FAR re-reference
        end if
    end if
end function
```

---

iv. Assumptions and Improvements to the existing algorithm

- We have made several assumptions and improvements over the course of the program.
- Firstly, the SHiP/SRRIP paper suggests searching for the victim with the highest interval from the head of the cache set. But we have observed that such a search will lead to low performance as we might be removing the cache line which was in the MRU position of the set. We modify the algorithm to start the search from 13th LRU position of the set so that the 12 MRU lines in the cache will essentially contain the line with the highest probability of getting a HIT.
- Secondly, the algorithm recommends using the FAR interval to be DISTANT interval – 1 but we have changed it to -2 as the change might force some of the blocks to stay longer in the chain and benefit from the learning rather than being quickly evicted.

- Third, the original SHiP paper [2] recommends using 3-bit counter members and 2-bit wide intervals. However, we have increased the counters to 6 bits and intervals to 3 bits to finetune the performance.
- Fourth, we have changed the algorithm where we increment the distance intervals by 1 when the victim is not found. What we have done is, for the immediate and near immediate intervals we increment the counter by 1 and for the FAR/Near-FAR entries we increment the RRPV by 2, so that we will increase the speed of the predictor as well as flush the lines with least probability of getting a hit.
- Fifth, during the implementation of SRRIP the literature [1] says that we can pick sampler sets to predict the behaviour of the entire cache. However, we have chosen not to pick the sampler sets as it will simply increase the complexity of the code at the cost of reduced performance. Moreover, even with associating the signature with every cache line, we are still within the maximum usage allowed for the project.
- Sixth, during UpdateMyPolicy, we have computed signature in such a way that, it will index a separate counter table for each type of access.
- Finally, we are updating the position of the entries to install it at the MRU position if the entry has the Re-reference interval of less than FAR value. This will make sure that the high probable blocks are saturated at the MRU and low ones are at the LRU positions.

Simulation Evaluation:

i. Challenges faced:

- We faced several challenges during the coding and debugging the environment. The SHiP paper is ambiguous when it comes to finding the victim for eviction. We tried several approaches to figure out what that is, once we read the RRIP [1] paper, it became clear that the SHiP doesn't have an inbuilt victim selection policy rather it exploits the SRRIP for the victim selection.
- There was a bug in our implementation of SRRIP. We weren't updating the intervals on the cache hits as recommended by FP/HP policies in the literature [1]. After reading the algorithm of SRRIP again, it became clear that we need to reduce the RRPV values of the hit blocks so that their eviction probability reduces. We chose frequency priority for the update policy.
- Understanding the infrastructure was challenging as it uses multiple class level method calls to communicate the prediction target/direction. We used gdb and printf extensively to understand the infrastructure.

ii. Latency:

We have exploited several features of C and C++ to reduce the prediction latency and heap usage:

- memset: The initialization process of the counters is tedious and time-consuming especially when we are using many counters for the high prediction accuracy. We have used memset in C++ which is much faster in setting the initial value of the counter tables.
- During the victim capture loop, it will take a few more cycles if we are incrementing the distance value by 1, when the victim is not found. We enhanced the process by incrementing the counters by 2 for the distances that are not in the immediate reference position.

iii. Results:

We present the results of our project that shows the effectiveness of our proposal. We conducted

experiments on different parameter settings and found that for the counter width of 6 bits, interval width of 3 bits and counter table size of 18 bits we are getting the best possible values as seen in the below set of figures.

Conclusion:

Because LLC reference patterns are filtered by higher-level caches, typical spatial and temporal locality patterns are much harder to optimize for. Here, we have implemented a simple and effective approach called SHiP++ for predicting re-referencing behavior for LLCs. SHiP++ enhances SHiP re-reference predictions on cache insertions and cache hits, improves SHCT training, and finally makes cache-access type aware re-reference. The predictor gives substantial IPC improvements over the existing LLC cache predictors. The programming language is C++ and we have made several enhancements to improve the predictor's accuracy. Experimental results showed that it performs equal to or better than LRU for a set of 16 traces out 27 distributed SPEC 2006 traces as seen from the figure 3. We have also presented an exhaustive test of the LLC predictor built on different counter sizes. As we can see from figure 4 with the increase in the counter hardware budget, it can achieve even better performance, e.g. 2.62 Geometric Mean IPC under very-high 256 KB budget vs 2.525 IPC with 4KB budget. Overall, SHiP++ solves the weaknesses of SHiP and improves performance predictions and provides 1.8% speedup over SHiP and 6.5% speedup over LRU (Figure 5).

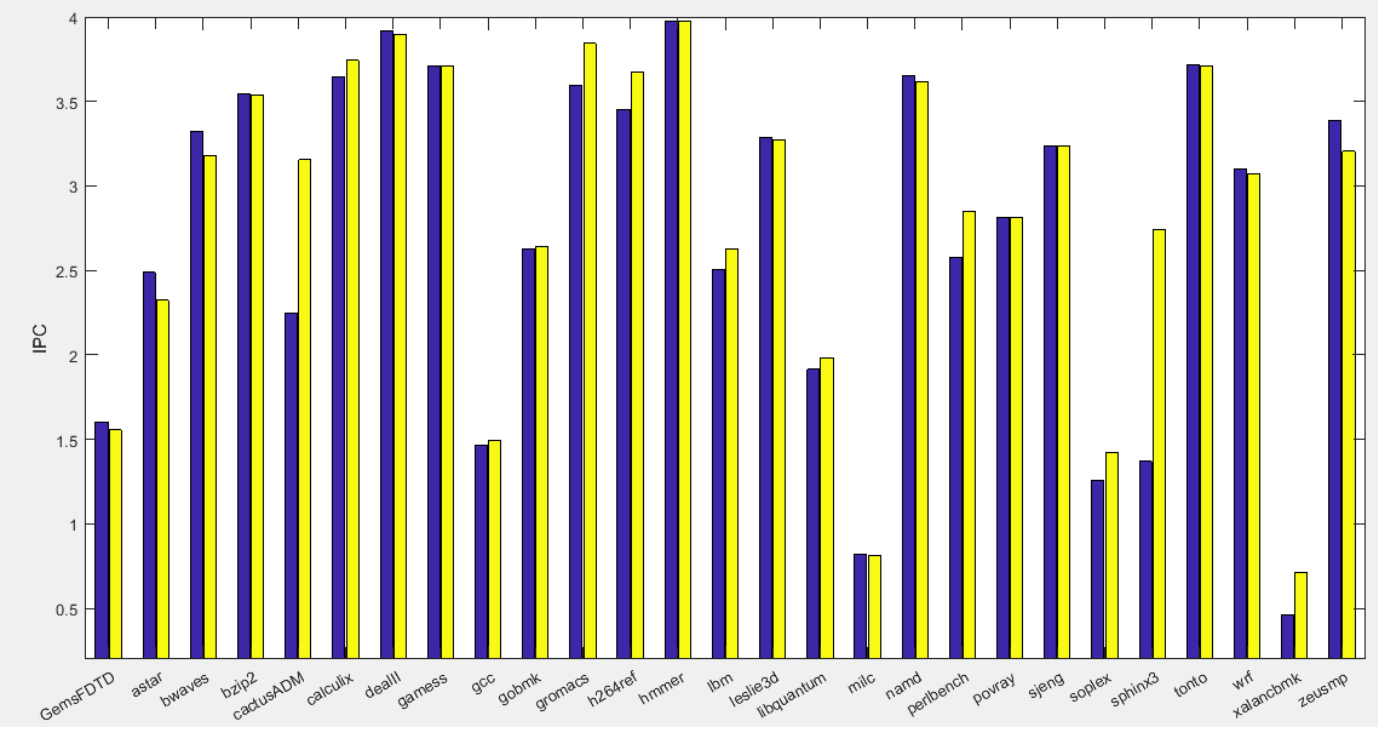

Fig 3. Average IPC of SHiP++(yellow) vs LRU(blue) for the set of distributed SPEC traces.

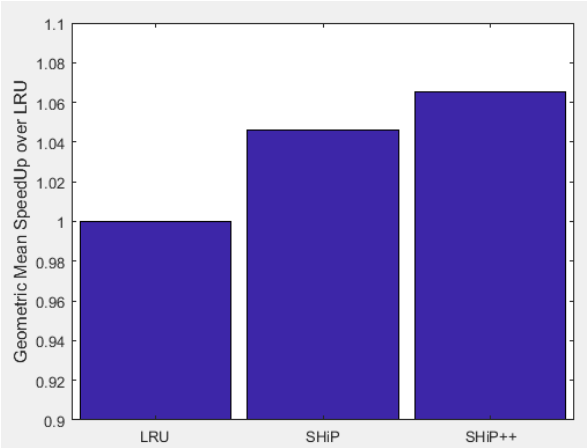

Fig 4. G.M. Speedup of LRU vs SHiP vs SHiP++

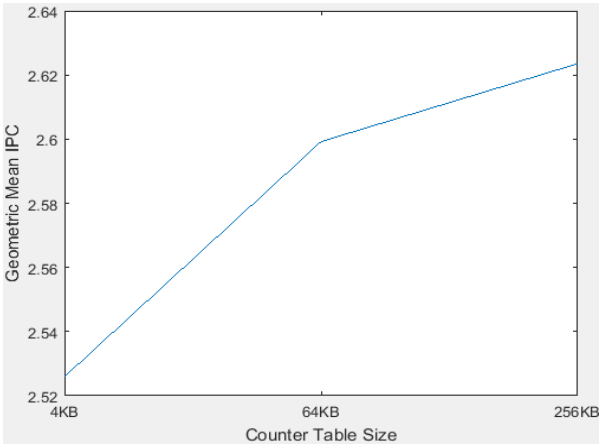

Fig 5. Mean IPC of SHiP++ with different SHCT Counter Sizes

References:

[1] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” in ACM SIGARCH Computer Architecture News, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[2] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, “Ship: Signature-based hit predictor for high performance caching,” in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011, pp. 430–441.

[3] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moinuddin Qureshi, “SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance”, CRC2, 2017

[4] S. M. Khan, Y. Tian and D. A. Jimenez, "Sampling Dead Block Prediction for Last-Level Caches," 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, 2010, pp. 175-186.

[5] Elvira Teran, Zhe Wang, and Daniel Jimenez. 2016. Perceptron Learning for Reuse Prediction. Int’l Symp. on Microarch. (2016).