# SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance

Vinson Young vyoung@gatech.edu
Chia-Chen Chou cchou34@gatech.edu
Aamer Jaleel ajaleel@nvidia.com
Moinuddin Qureshi moin@ece.gatech.edu

## ABSTRACT

In this paper, we analyze the state-of-the-art replacement policy based on *Signature-Based Hit Prediction (SHiP)*. While we observe that the proposed SHiP implementation improves performance over the LRU policy on all configurations, we identify three areas of improvement. First, the policies for updating the counters in the *Signature History Counter Table (SHCT)* can be modified. Second, a broader range of insertion points for an incoming line can be used based on the value of the SHCT counter. Third, cache performance under prefetching can be enhanced by using a separate SHCT table for demand and prefetch lines and modifying the insertion/promotion of prefetch lines. We call our proposed implementation of SHiP with these enhancements as *SHiP++*.

We evaluate SHiP++ on 17 benchmarks from the SPEC2006 suite. For the single-core without prefetching configuration, SHiP++ improves performance by 6.2% over LRU and 2.2% over SHiP. For the single-core with prefetching configuration, SHiP++ improves performance by 4.6% over LRU and 2.2% over SHiP. SHiP++ incurs an overhead of less than 20KB for managing the 2MB cache, well within the provisioned 32KB.

## 1. INTRODUCTION

Computer designers look for techniques that can improve performance, increase energy-efficiency, and reduce the off-chip bandwidth bottleneck. Quite often, a solution that improves one of these metrics ends up worsening the other metrics. However, an intelligent cache replacement policy is one such optimization that can simultaneously provide all three benefits: it can improve system performance by reducing the long-latency memory accesses, increase energy efficiency by servicing data on-chip at lower energy, and mitigate the pressure on the off-chip memory systems. While the commonly used LRU (Least Recently Used) replacement policies works well for L1 caches, it tends to be not as effective for the Last Level Cache (LLC). This is because the LLC receives an access stream that has filtered temporal locality. Over the last decade, there has been significant research work on intelligently managing the LLC by accurately predicting the re-reference interval of cache lines. We briefly describe two prior proposals that form the basis of our design: *Re-Reference Interval Prediction (RRIP)* [1] and *Signature-Based Hit Predictor (SHiP)* [4].

## 2. BACKGROUND: RRIP AND SHIP

The LRU policy always predicts a near immediate re-reference interval on cache hits and misses. However, such a prediction causes poor performance for applications that have frequent bursts of references to non-temporal data (often called *scans*). The RRIP [1] policy is designed to protect the cache from scans. 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)*.

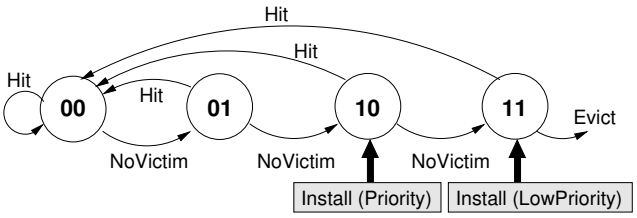


**Figure 1: Re-Reference Interval Prediction (RRIP) .**

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.

Signature-Based Hit-Predictor (SHiP) [2, 4] is a subsequent proposal that tries to predict the reuse characteristics of a line based on the signature of the line (gathered by the miss-causing PC, or memory region, or the instruction sequence leading to the load). The reuse characteristic of a line is heavily correlated based on the Program Counter (PC) of the instruction that caused the miss. Some PCs tend to access lines that are heavily reused and others access lines that remain unused. If we know the past behavior of the PC then we can predict the likely reuse characteristic of the incoming line. Figure 2 shows the organization of SHiP. The key element of SHiP is the *Signature History Counter Table (SHCT)* that contains a counter (SHCTR) for each signature. Each line (of a few sampled sets) is appended with its signature (14-bit hash of the miss-causing PC) and a reuse (*R*) bit. To reduce storage, only 32-64 sampled sets are used.

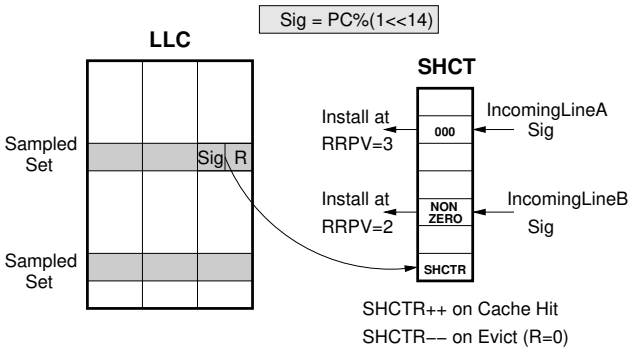


**Figure 2: Signature-Based Hit Prediction (SHiP).**

When a line is installed in the cache, the *R* bit associated with that line is set to zero. If the line receives a hit, the *R* bit of the line is set to 1 and the SHCTR corresponding to the signature of the line is incremented (in a saturated manner). When the line gets evicted and the *R* bit is zero, the SHCTR associated with the line signature is decremented (in a saturated manner). The incoming line is installed with RRPV of 3 if the SHCTR associated with the signature of the incoming line is zero, and with RRPV of 2 otherwise. Note that the SHiP design, as proposed, does not differentiate between prefetch and demand accesses.

## 3. DESIGN: ENHANCING SHIP TO SHIP++

While SHiP performs well by learning the re-reference characteristic of PCs, we enhance SHiP re-reference predictions at cache insertion and cache hits. Furthermore, we also enhance the training for the Signature History Counter Table (SHCT) on cache hits. Finally, we propose *cache access type aware* re-reference predictions rather than using a single re-reference prediction for all cache access types. We discuss these enhancements now.

**Enhancement 1: Improved Cache Insertion**. SHiP learns the re-reference pattern of PCs using the SHCT and predicts the likelihood of a re-reference based on the SHCT value. Specifically, when the SHCT value of a given PC is saturated at its minimum value (i.e., zero), SHiP learns that insertions made by this PC are rarely re-referenced, and thus inserts all lines with RRPV=3. When the SHCT value is non-zero, the baseline SHiP policy inserts all lines with RRPV=2 and attempts to *learn* the re-reference behavior of the line. We enhance SHiP by realizing that when the SHCT value of a given PC is saturated at its maximum value (seven in our studies), there is already strong evidence that insertions made by this PC are often re-referenced. As such, rather than re-learning the re-reference behavior upon every insertion, such lines can directly be inserted with RRPV=0. This enhancement retains useful lines for a longer duration in the cache.

**Enhancement 2: Improved SHCT Training**. SHiP trains the SHCT table on cache hits and cache evictions. We propose improvements to the SHCT on cache hits. While SHCT updates on cache hits (rather than cache evictions) enable SHiP to train re-reference behavior quickly, SHiP trains cache hits disproportionately to cache evictions. For example, cache hits to the same line constantly updates the SHCT. We find that training only on the first re-reference of a line can train the table more effectively by weighting cache hits and cache evictions similarly.

**Enhancement 3: Writeback-Aware RRPV Updates**. In general, writeback requests are a side effect of organizing a write-back cache hierarchy (rather than a write-through hierarchy). Writeback requests are non-demand background cache requests to update the contents of the larger levels of a cache hierarchy. Such requests provide no indication of reuse, and tend to signify the end-of-use of the line in the hierarchy. As such, we propose to statically predict RRPV=3 for all writeback insertions. In doing so, we insert writebacks low-priority so they do not utilize valuable cache space for a long period of time.

**Enhancement 4: Prefetch-Aware SHCT Training**. SHiP does not distinguish between different types of cache accesses and applies a single re-reference prediction. Specifically, in a system with prefetching, the re-reference behavior of demand accesses and prefetch accesses can be quite different [5]. For example, an aggressive prefetcher can pollute the cache by bringing in cache lines that are never re-referenced. In such systems, it would be beneficial to independently measure the re-reference behavior of demand accesses and prefetch accesses. Thus, we propose to assign different signatures for demand and prefetch accesses. We implement this by appending prefetch information into the signature ($signature = (PC << 1) + is\_prefetch$). As such, the SHCT independently learns the re-reference behavior of prefetch and demand requests.

**Enhancement 5: Prefetch-Aware RRPV Updates**. SHiP always updates RRPV to near-immediate (i.e., RRPV=0) on all cache hits. We propose different RRPV update policies for prefetch and demand requests on cache hits. We modify RRPV updates for prefetch requests in two different scenarios. The first scenario is when a prefetched line is subsequently re-referenced by a demand request. In such a situation, SHiP incorrectly updates the RRPV to zero. For example, consider a streaming workload where an aggressive prefetcher hides latency by prefetching lines subsequently consumed by demand requests. In such a workload, a line receives no subsequent re-references after the demand request. We improve RRPV updates for such workloads by de-prioritizing prefetched cache lines on demand accesses [3]. Specifically, when a demand access hits on a line that was brought in by a prefetcher, we always update the RRPV to '3' rather than '0'. To do so, We propose one bit per cache line to remember whether a line was inserted with a demand access or a prefetch access using a *is_prefetched*-bit. When a demand access hits a prefetch line *is_prefetch*-bit=1, we unset the prefetch-bit and set RRPV=3. Note that subsequent demand or prefetch requests to such a line update the RRPV to zero. It is only the first demand access to a prefetched line that downgrades the RRPV=3. The second scenario is when a prefetched line is subsequently re-referenced only by a prefetch request. Such situations occur when the workload working-set is primarily serviced by prefetchers. Since these workloads are prefetch-friendly, we propose to not give high priority to such cache lines and not update the RRPV (i.e, keep RRPV at its current value).

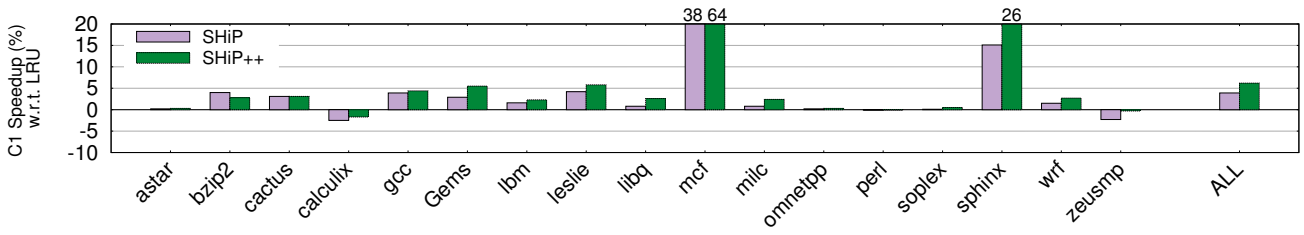


**Figure 3: Performance of SHiP++ for the single-core configuration without prefetcher (`c1`).**

## 4. STORAGE OVERHEAD ANALYSIS

We analyze the storage overhead of SHiP++ in four components: (1) Per-line metadata, (2) Storage for the Signature History Counter Table (SHCT), (3) Extra metadata stored for the sampled sets, and (4) Storage to identify the sample sets.

The per-line metadata includes re-reference-interval bits (RRPV), which is 2 bits for every line. For configurations with prefetches, every line is equipped with a *is_prefetched* bit that indicates whether the line was installed by a prefetch.

SHCT has 16K entries, each a 3-bit counter. For multi-core systems, each core has a private SHCT. Therefore, SHCT incurs 6KB for single-core and 24KB for quad-core.

We use 64 sampled sets in our design. Each line in the sampled set contains a 14-bit signature and a Reuse (R) bit, so 15 bits per line. For the multicore configurations, we also track the *core_id* of the core that installed the line (in order to properly update the private SHCT of that core). Therefore, each line in the sampled set would have 15-17 bits of metadata. Given a 16-way cache, 64 sampled sets, would have 1K lines. Therefore, the storage overhead of the sampled sets is only 15Kbit-17Kbit (1.875KB-2.125KB).

To identify the sampled sets, we assume a 64-entry table that keeps track of the sets that are identified as sampled set. With 2-byte per entry to identify the set, this structure would require 128 bytes. Table 1 shows the total storage of all configurations. Single-core configurations at most 20KB and quad-core configurations use less than 93KB, both within the competition budget of 32KB and 128KB, respectively.

**Table 1: SHiP++ Storage Requirements**

| | Config 1 | Config 2 | Config 3 | Config 4 |
|---|---|---|---|---|
| RRPV (2-bit/line) | 8KB | 8KB | 32KB | 32KB |
| *is_prefetched* (1-bit/line) | 0 | 4KB | 0 | 16KB |
| SHCT (16K-entry/core) | 6KB | 6KB | 24KB | 24KB |
| Sampled sets (64 sets) sig + reuse (+core_id) | 1.875KB | 1.875KB | 2.125KB | 2.125KB |
| SampleSet ID (64 entries) | 128B | 128B | 128B | 128B |
| Total Storage Overhead | 16KB | 20KB | 76.25KB | 92.25KB |

## 5. EXPERIMENTAL METHODOLOGY

We use ChampSim CRC2 Simulator to evaluate our cache replacement policy. The four specified configurations include single-core and multicore configurations, with and without prefetching, shown in Table 4. We perform evaluations on all of SPEC suite that has at least 0.5 MPKI (17 workloads). We warm up the cache for 10 million instructions and then perform timing simulation until each benchmark in a workload executes at least 100 million instructions. We report the instruction per cycle (IPC) as the performance metric. All numbers are normalized to the baseline LRU policy, and we use geometric mean for reporting the average speedup.

**Table 2: CRC2 Configurations**

| | Config 1 | Config 2 | Config 3 | Config 4 |
|---|---|---|---|---|
| Core Count | 1 | 1 | 4 | 4 |
| LLC | 2MB | 2MB | 8MB | 8MB |
| Prefetcher | None | L1/L2 | None | L1/L2 |

## 6. RESULTS

We present the results for single-core tracks C1 and C2 and provide analysis that shows the effectiveness of our proposal. Due to time constraints, we have not optimized our proposal for multi-core configurations, but provide the code in our submission for testing.

### 6.1 Results for Single Core without Prefetcher

The first configuration is single core without prefetcher (referred to as `c1`). Figure 3 compares the performance of SHiP++ to those of the baseline LRU and the original SHiP. The speedup is with respect to LRU, and the right bar is the geometric mean of all single-core workloads. Unlike SHiP that always inserts lines at low priority, SHiP++ inserts high-confidence lines at highest priority, which improves performance for *calculix*, *Gems*, and *zeusmp*. Moreover, SHiP++ outperforms SHiP for *mcf* and *sphinx* by training SHCT only once per line re-use, which avoids over-training on constantly used lines. On average, SHiP outperforms the baseline by 3.8%, but SHiP++ outperforms the baseline by 6.2% with a maximum speedup of 64%.

The figure of merit for evaluating the systems for the competition is the geometric mean of the individual IPC. Table 3 reports the geometric mean of the 17 workloads for LRU, SHiP, and SHiP++. The GeoMean IPC of LRU is 0.507. SHiP increases this to 0.526, thus providing 3.9% improvement. SHiP++ increases the GeoMean to 0.538, thus providing 6.2% improvement over LRU.

**Table 3: GeoMean IPC for (`c1`) Configuration**

| Policy | Absolute GeoMean IPC | Normalized GeoMean IPC |
|---|---|---|
| LRU | 0.507 | 100.0% |
| SHiP | 0.526 | 103.7% |
| SHiP++ | 0.538 | 106.1% |

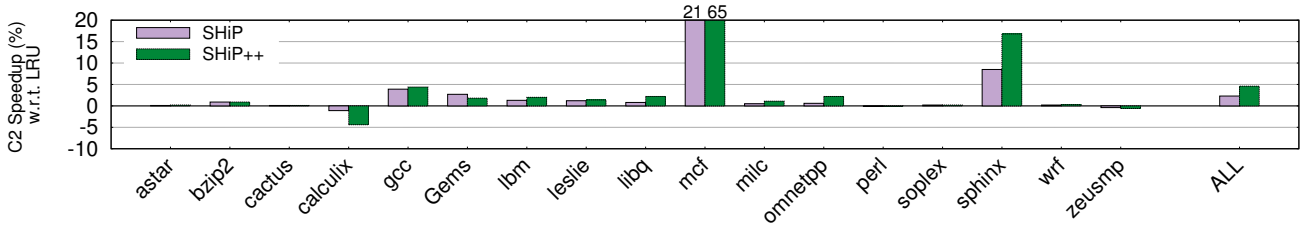


**Figure 4: Performance of SHiP++ for the single-core configuration with prefetcher (c2).**

## 6.2 Results for Single Core with Prefetcher

For second configuration (single core with prefetcher, c2), SHiP++ learns prefetch usefulness separately from demand usefulness, by using half the of SHCT table for prefetch and half the SHCT table for demand. In addition, SHiP++ protects demand requests against prefetch requests, by not updating state for prefetched lines and demoting prefetched lines to RRPV=3 on demand access. Figure 4 shows that for workloads such as *sphinx* and *mcf*, SHiP++ learns that their prefetches are unlikely to be re-used and should be installed at low priority. Overall, SHiP improves performance by 2.4%, and SHiP++ provides a speedup of 4.6%.

The figure of merit for evaluating this configuration for the competition is also the geometric mean of the individual IPC. Table 3 reports the geometric mean of the 17 workloads for LRU, SHiP, and SHiP++. The GeoMean IPC of LRU is 0.663. SHiP increases this to 0.678, thus providing 2.4% improvement. SHiP++ increases the GeoMean to 0.694, thus providing 4.6% improvement over LRU. Thus, our enhancements of SHiP has almost doubled the performance benefit obtained by the original SHiP implementation.

**Table 4: GeoMean IPC for (c2) Configuration**

| Policy | Absolute GeoMean IPC | Normalized GeoMean IPC |
|---|---|---|
| LRU | 0.663 | 100.0% |
| SHiP | 0.678 | 102.4% |
| SHiP++ | 0.694 | 104.6% |

Figure 5 shows the distribution of MPKI for all 17 workloads (the rightmost bar is the average). The MPKI is divided into four categories: Load, RFO, Prefetch, and Writeback. We observe that for a system under prefetching, a majority of the cache accesses and misses are caused by the prefetcher. Therefore, to improve cache management, it is important to pay attention to prefetch requests (and installs).

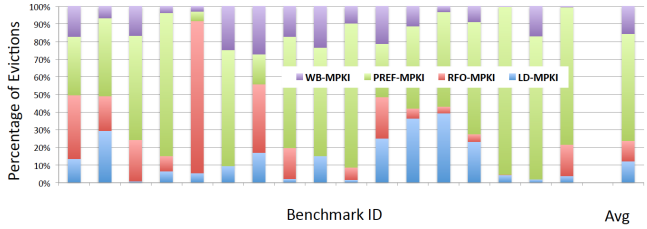


**Figure 5: The MPKI distribution for the C2 configuration: access types include Load (LD), Request for Ownership (RFO), Prefetch (PREF), and Writeback (WB).**

# 7. SUMMARY

While SHiP is effective at improving last-level cache hit rate, it has several aspects which can be further improved. We propose SHiP++ which enhances SHiP re-reference predictions on cache insertions and cache hits, improves SHCT training, and finally makes cache-access type aware re-reference predictions. In particular, SHiP++ improves:

1. Insertion position: SHiP++ inserts at highest-priority RRPV=0 for lines with the highest confidence of re-use.
2. SHCT training policy: SHiP++ trains only on first use of line to avoid over-training from heavily used lines.
3. Writeback updates: SHiP++ inserts all writebacks installs at lowest-priority (RRPV=3)
4. SHCT training on prefetches: SHiP++ uses a separate dedicated SHCT to learn the reuse behavior of prefetched lines, and thus avoids the interference of reuse in demand from reuse in prefetch.
5. Prefetch updates: Prefetches are either demoted on demand hit or kept the same on prefetch hits, so as to give low priority to lines that are easily prefetchable.

Overall, SHiP++ solves the weaknesses of SHiP and improves performance over multiple configurations. For single-core configuration, SHiP++ provides 2.2% speedup over SHiP and 6.2% speedup over LRU. For single-core with prefetching, SHiP++ provides 2.4% speedup over SHiP and 4.6% speedup over LRU.

# 8. 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] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 737–749. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037701

[3] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 51:1–51:22, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2677956

[4] 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.

[5] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 442–453.