Ranking-Glider : Learning the victim of cache replacement

Nayeon Lee
School of Computing, KAIST
Daejeon, South Korea
nlee0212@kaist.ac.kr

Taeyoung Kim
School of Computing, KAIST
Daejeon, South Korea
taeyoungkim21@kaist.ac.kr

Abstract

Cache replacement policy was usually heuristic-based, but state-of-the-art solutions are taking a learning-based approach that learns from past behaviors to predict future replacement policy. Most such works model the cache replacement problems as binary classification, classifying whether an incoming lane is cache-friendly or cache-averse. We suggest a different view of this problem, which compares the incoming and existing cache lanes to choose the victim lane. Using this point of view, we modify the recently suggested learning-based policy to learning to rank and evaluate our implementation of the ChampSim-based simulator used in the CRC-2 championship. Our model results in similar hit rates with other policies while outperforming for a few cases.

1 Introduction

Exploiting caches play a significant role in decreasing the latency of memory accesses. It is necessary to have a cache replacement strategy to select a cache block to be replaced on a cache miss due to limited cache memory. However, the cache replacement policy is known to be challenging to construct. The main target of the policy is to pick *which* cache block should be substituted, which is more sophisticated than binary problems. On top of that, the pattern of usage of data and instructions varies significantly among different tasks, and the task themselves are complicated. As a solution, learning-based solutions were proposed and showed state-of-the-art performance.

In this paper, we propose Ranking-Glider, a novel cache replacement policy built upon a learning-based solution, Hawkeye [1], and a machine learning-based solution, Glider [2]. Our proposed policy exploits Ranking SVM, which typically learns to rank to select the cache block to be replaced on a cache miss. We suggest two methods: (1) maintaining a single Ranking SVM (R-Glider); (2) building a Ranking SVM for each of the PCs (R-Glider-PC). We compare the two solutions with existing policies on different configurations and datasets. The results of the experiments show that our solution achieves comparable performance with other policies and even outperforms them in a few cases.

2 Related works

Previous solutions fall into two categories, heuristic-driven solutions like LRU or MRU and learning-based solutions. Learning-based solutions like SDBP [3] or SHiP [4] try to mimic the behavior of heuristic-driven solutions and learn whether a current memory instruction is cache-friendly or cache-averse.

Our work builds upon Glider [2] which utilizes Hawkeye [1], so we give a detailed explanation of these two works.

2.1 Hawkeye

Hawkeye changes the viewpoint of this learning problem as supervised learning with optimal solution generated from Belady's algorithm [5]. The core idea of this is, among the cache hits, if we separate them by the term between accesses, heuristics-based approaches like LRU and DRRIP fail to capture medium-term or long-term reuse. Learning-based solutions SDBP and SHiP can capture the medium-term reuse but no long-term reuse. However, Belady's algorithm for the optimal solution picks up all three kinds of cache hits. So Hawkeye proposes to generate the optimal solution with a simple algorithm named OPTgen, which computes suboptimal solutions given a limited view of the future access. Given this approach, Hawkeye uses a simple predictor that maintains 3-bit counters for each 13-bit hashed PC, which is trained positively or negatively based on OPTgen's result and uses the most significant bit as a determination of cache-friendliness.

Hawkeye contains several details in their implementation adopted by Glider and our R-Glider. For example, generating the optimal solution from OPTgen for all the sets in the cache creates a significant bottleneck in cache access. Hence Hawkeye only tracks 64 sets from 2048 sets and trains their predictor only on these sets. Since the input to the predictor is the PC of store/load operation, the training on the subset of cache sets can be generalized to others. In the more recent version of Hawkeye [6] whose implementation was the starting point of our project, the authors observed that the cache behavior is different between on-demand fetches and prefetchers. Based on this observation, they suggest using two predictors, one for on-demand fetch prediction and prefetching fetch prediction. Based on the idea that other learning-based cache replacement policies can leverage the observation, we used this feature in our implementation by maintaining two Ranking SVMs.

2.2 Glider

Built on Hawkeye's solution, Glider replaces the predictor of Hawkeye based on the observation from a deep learning-based policy. By training an attention-based neural network on OPTgen, the authors found that only a few source memory accesses influence the caching decision of the target memory access. They also observed that the PC history is a powerful feature for learning the result, while their order is irrelevant. From this observation, authors use the k-sparse PC history as the feature of SVM. For example, if the most recent PCs are $P_2, P_4, P_7, P_{10}, P_{13}$, the output of SVM is defined as

$$w_2 + w_4 + w_7 + w_{10} + w_{13}.$$

They maintain an Integer SVM for every load/store PC, so having N load/store instructions require N SVMs.

Since a floating-point computation becomes a considerable bottleneck for architectures, Glider instead uses Integer SVM, which constrains all the parameters to be an integer and the learning rate to 1.

3 Proposal

Glider uses Integer SVM, which maintains the weights w_i for each PC P_i to compute the output of PC history $P_{i_1}, \ldots, P_{i_k}$ as

$$f(P_{i_1}, \ldots, P_{i_k}) = w_{i_1} + \ldots + w_{i_k}.$$

In our implementation, we replace this Integer SVM with an Integer Ranking-SVM, which instead receives two inputs and rank them as

$$f(P_{i_1}, \ldots, P_{i_k}, P_{j_1}, \ldots, P_{j_k}) = w_{i_1} + \ldots + w_{i_k} - w_{j_1} - \ldots - w_{j_k}. \quad (1)$$

Here, the $P_{i_1}, \ldots, P_{i_k}$ correspond to current PC history and $P_{j_1}, \ldots, P_{j_k}$ refers to the PC history of the cache lines inserted into cache.

We used the Hawkeye implementation as the starting point of our project. We obtained the source code for it from [7] [1] and the implementations of other baselines like LRU and SRRIP given by CRC-2's example. Our repository `https://github.com/nlee0212/RankingGlider` includes these baselines along with R-Glider source code.

We first implemented the Ranking Integer SVM, which maintains one signed integer for each load/store instruction PC. Since we use a small length of PC history, We implemented the weights as

[1] The source is currently not-available.

```
int32_t max_prediction = -2147483648;
int32_t curr_prediction;
int32_t lru_victim = -1;
for (uint32_t i=0; i<LLC_WAYS; i++) {
  if (prefetched[set][i])
    curr_prediction = prefetch_predictor->get_value(curr_pc_hist, cache_pc_hist[set][i]);
  else
    curr_prediction = demand_predictor->get_value(curr_pc_hist, cache_pc_hist[set][i]);
  if (curr_prediction > max_prediction) {
    max_prediction = curr_prediction;
    lru_victim = i;
  }
}
for (uint32_t j=0; j<k; j++) {
  victim_pc_hist[j] = cache_pc_hist[set][lru_victim][j];
  cache_pc_hist[set][lru_victim][j] = curr_pc_hist[j];
}
if( SAMPLED_SET(set) && max_prediction < 0){
    if(prefetched[set][lru_victim])
        prefetch_predictor->decrement(curr_pc_hist,victim_pc_hist);
    else
        demand_predictor->decrement(curr_pc_hist,victim_pc_hist);
}
return lru_victim;
```

Figure 1: Code for getting victim in set, LRU-like.

```
  void decrement (uint64_t curr_pc_hist[hist_n], uint64_t victim_pc_hist[hist_n])
  {
    for (int i = 0; i < hist_n; i++) {
      uint64_t curr_sig = CRC(curr_pc_hist[i]) % SHCT_SIZE;
      if (TABLE.find(curr_sig) == TABLE.end())
        TABLE[curr_sig] = 0;
      if (TABLE[curr_sig] != min_weight)
        TABLE[curr_sig]--;

      uint64_t victim_sig = CRC(victim_pc_hist[i]) % SHCT_SIZE;
      if (TABLE.find(victim_sig) == TABLE.end())
        TABLE[victim_sig] = 0;
      if (TABLE[victim_sig] != max_weight)
        TABLE[victim_sig]++;
    }
  }
```

Figure 2: Code for training Rank-SVM negatively on current PC history.

a dynamic Map instead of a fixed array of weights since the number of PC is unknown at the initial point. [2]

Now for each block of cache, we add length k vector of PC, which will be used as $P_{j_1}, \ldots, P_{j_k}$ used in (1). When we insert a new block to the cache, we update this length k vector with the current PC history.

With these Ranking-SVM and feature vector of cache, we modify the cache access as follows. Note that all the training of SVMs uses a total of 2048 sets, each with 64 sets like Hawkeye, and we maintain two separate Ranking-SVMs for on-demand fetch and prefetch.

- When finding the victim, for each block in the K-way, we get the PC history of each block in the cache and rank it with the current PC history. Two options for finding the victim are, using the block with the smallest f value defined in (1) (which is similar to LRU) or using the first block having $f < 0$ (which is similar to RRIP).
- After selecting a victim cache block, we train the Ranking-SVM negatively on the current PC history.
- Behind each cache fill, we generate the OPT solution for past instructions from OPTgen and train the Ranking-SVM on the result of OPT solution.

Parts of the code implementations can be seen in Figures 1 and 2. We implemented another version of R-Glider that contains Ranking-SVM for each PC, which is R-Glider-PC. The only difference is that a specific Ranking SVM for the current PC is trained and used for prediction.

[2] We are curious how this feature can be implemented at the architecture level, where it was available since our implementation was on the C++ of the simulation framework. Since Glider uses the same approach, we just decided to use it.

3.1 Hardware budget

In this section, we analyze how much memory our implementation uses. We consider a 16-way 2MB cache, with 2048 PCs and $k = 5$ for the sparse feature.

Since we use the same replacement state, sampler, and OPTgen for the Hawkeye and Glider, these account for 12KB, 12.7KB, and 4KB. Now we contain at most 2048 weights for each SVM, and the weights are 8-bits wide, so each SVM takes at most 2KB. We also track the PC history with five 8-byte for each cache line and the current PC history, increasing 1.2MB to the hardware budget. This PC history for each cache line holds most of the portion of our hardware budget.

4 Experiment

4.1 Experiment details

ITLB and DTLB	16-set, 8-way, 4KB
2nd-level TLB	128-set, 12-way, 96KB
Private L1 instruction/data cache	64-set, 8-way, 32KB
Private L2 cache (unified)	512-set, 8-way, 256KB
Shared LLC	2048-set, 16-way, 2MB for single core 8192-set, 16-way, 8MB for 4-core

Table 1: Cache hierarchy models

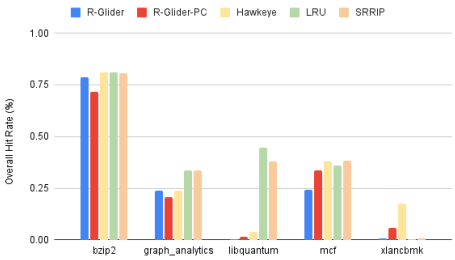
We evaluate two versions of Ranking-Glider (R-Glider and R-Glider-PC) using the simulation framework released by the 2nd Cache Replacement Championship (CRC-2) [8] based on Champsim simulator [9]. Table 1 contains the cache hierarchy model detail. Our experiments use four different configurations: (1) Single-core without a prefetcher; (2) Single-core with L1/L2 data prefetchers; (3) A 4-core configuration without a prefetcher; (4) A 4-core configuration with L1/L2 data prefetchers. We refer to these configurations as config(1), (2), (3), and (4), respectively, throughout the paper. The code provided [7] contains further details on the baseline model.

We performed experiments upon some of the SPEC 2006 benchmarks and CloudSuite 2.0 benchmarks. CloudSuite 2.0 benchmarks contain traces with a single task distributed among four cores. Figures 3 and 4 show hit rates for the SPEC 2006 benchmarks in a single core and a 4-core setting, and Figure5 show hit rates for the CloudSuite 2.0 benchmarks in a 4-core setting. Figure 6 shows IPC on all of the benchmarks given in a 4-core setting with a prefetcher. We compare the performance with other solutions, which are Hawkeye [1], LRU, and SRRIP [10]. We use a sample of 10 million instructions for each benchmark and evaluate our solution with 1 million warm-up instructions and 10 million simulation instructions. The trace files were available initially in the modified version of ChampSim provided at CRC-2 [8].

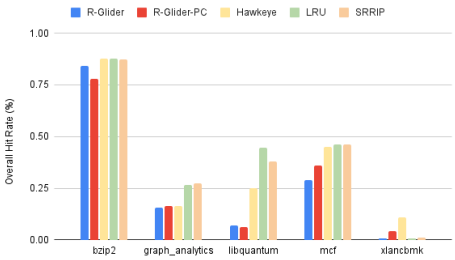
4.2 Experiment results

Figure 3 shows that R-Glider and R-Glider-PC in a single core setting rely heavily on the dataset used. R-Glider shows comparable results on some datasets, but in others, it does not. Also, the usage of Ranking SVM for each of the PCs seems ineffective for all datasets. However, in Figure 4, R-Glider-PC tends to outperform R-Glider in a 4-core setting with different tasks performed on each core. It is also noticeable that R-Glider or R-Glider-PC exceeds Hawkeye, LRU, or SRRIP when using specific datasets. From this, we can conclude that in a 4-core setting with different tasks allocated in each core, it is usually beneficial to have one Ranking SVM for each of the PCs so that each machine can keep track of relevant PCs for each of the PCs.

Figure 5 shows the average hit rate in a 4-core setting on each task distributed among four cores. It is shown in Figure 5a that the performance of R-Glider and R-Glider-PC relies on the datasets when there is no prefetcher. They also tend to show lower hit rates compared to other solutions. However, as shown in Figure 5b, when prefetchers exist, R-Glider outperforms R-Glider-PC on all datasets and sometimes outperforms other solutions in some datasets. We can then conclude that in a multi-core setting with a single task distributed among different cores, it is beneficial to use a single Ranking SVM for the whole program rather than allocate one Ranking SVMs for each of the PCs.

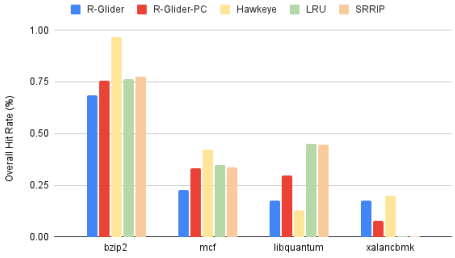


(a) hit rate on config(1)

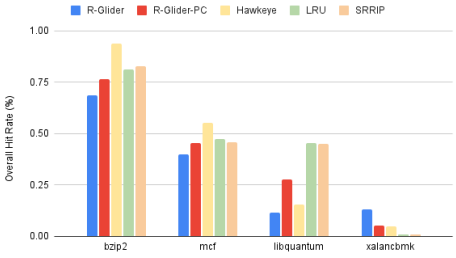


(b) hit rate on config(2)

Figure 3: Overall Hit Rate of SPEC 2006 benchmarks in a Single Core Setting

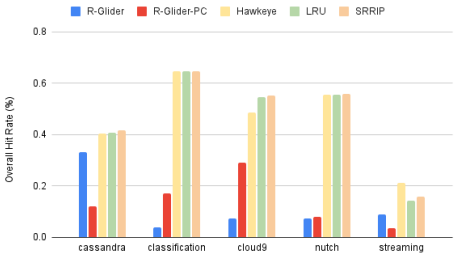


(a) hit rate on config(3)

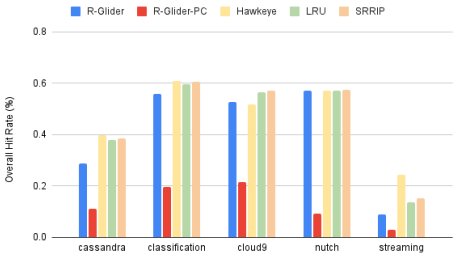


(b) hit rate on config(4)

Figure 4: Overall Hit Rate of SPEC 2006 benchmarks in a 4-Core Setting



(a) hit rate on config(3)



(b) hit rate on config(4)

Figure 5: Overall Hit Rate of CloudeSuite 2.0 benchmarks in a 4-Core Setting

Figure 6 shows the IPC in a 4-core setting with a prefetcher (i.e. config(4)). For the CloudSuite benchmarks, we used the average IPC among four cores for each dataset. IPC refers to instructions per cycle, which the lower, the better. As seen in the figure, the IPC value of R-Glider and R-Glider-PC does not increase significantly compared to other policies and even decreases in some cases. Even though R-Glider and R-Glider use a novel approach, the latency of memory access does not increase and even decreases sometimes, and, at the same time, the policies show comparable performances, proving that the proposed solution is quite competitive.

5 Conclusion

In this paper, we proposed a novel cache replacement policy that leverages Belady's algorithm and a machine learning model, Ranking SVM. We used Belady's algorithm to look back over a sufficiently long history of memory accesses to learn and mimic the optimal behavior [1], and used Ranking SVM(s) to rank PCs for cache replacement. We gained the idea of ranking PCs by starting from the observations provided in Glider [2], where it showed that target memory accesses are strongly correlated with just a few source memory accesses and that prediction accuracy is largely insensitive to the order of the sequence. We applied these observations by maintaining only k sequences of past PCs with no order information. We also gave out rank scores for each PC in a single Ranking SVM in one version, and constructing this Ranking SVM for each PC in another.

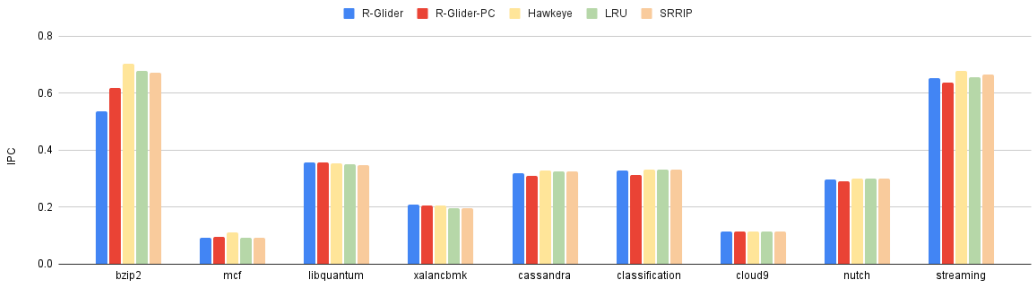


Figure 6: IPC on all benchmarks in a 4-Core Setting with a prefetcher

The experiments showed that the proposed policy achieved comparable performances in terms of hit rates and maintained similar IPC compared to existing solutions and previous state-of-the-art solutions. Based on the results, it is arguable that the proposed method is competitive.

However, more researches to take the cache replacement policy using Ranking SVM to the next level is necessary. Specific hyper-parameters such as k, the length of the PC history, minimum and maximum values for the rank scores, and the usage of RRPV values should be experimented in more detail with more experiments. We hope this paper can inspire other researchers to design a more efficient machine-learning-based cache replacement policy, specifically using Ranking SVM.

References

[1] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News*, 44:78–89, 06 2016. doi: 10.1145/3007787.3001146.

[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. ISBN 9781450369381. doi: 10.1145/3352460.3358319. URL `https://doi.org/10.1145/3352460.3358319`.

[3] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, 2010. doi: 10.1109/MICRO.2010.24.

[4] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5 (2):78–101, 1966. doi: 10.1147/sj.52.0078.

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

[7] A. Jain and C. Lin. Hawkeye cache replacement: Leveraging belady's algorithm for improved cache replacement, 2017. URL `https://crc2.ece.tamu.edu/?page_id=53`.

[8] Paul V. Gratz, Jinchun Kim, and Gino Chacon. The 2nd cache replacement championship, 2017. URL `https://crc2.ece.tamu.edu/`.

[9] Champsim: A trace-based cycle-accurate simulator, 2017. URL `https://github.com/ChampSim/ChampSim`.

[10] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH computer architecture news*, 38(3):60–71, 2010.