Yang HJ, Fang J, Cai M *et al.* A prefetch-adaptive intelligent cache replacement policy based on machine learning. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(2): 391-404 Mar. 2023. DOI: 10.1007/s11390-022-1573-3

A Prefetch-Adaptive Intelligent Cache Replacement Policy Based on Machine Learning

Hui-Jing Yang (杨会静), *Student Member, CCF*, Juan Fang* (方　娟), *Senior Member, CCF, ACM, IEEE*
Min Cai (蔡　旻), *Member, CCF*, and Zhi Cai (才　智), *Member, ACM*

Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China

E-mail: yangkx@emails.bjut.edu.cn; fangjuan@bjut.edu.cn; min.cai.china@bjut.edu.cn; caiz@bjut.edu.cn

Received May 10, 2021; accepted August 17, 2022.

Abstract Hardware prefetching and replacement policies are two techniques to improve the performance of the memory subsystem. While prefetching hides memory latency and improves performance, interactions take place with the cache replacement policies, thereby introducing performance variability in the application. To improve the accuracy of reuse of cache blocks in the presence of hardware prefetching, we propose Prefetch-Adaptive Intelligent Cache Replacement Policy (PAIC). PAIC is designed with separate predictors for prefetch and demand requests, and uses machine learning to optimize reuse prediction in the presence of prefetching. By distinguishing reuse predictions for prefetch and demand requests, PAIC can better combine the performance benefits from prefetching and replacement policies. We evaluate PAIC on a set of 27 memory-intensive programs from the SPEC 2006 and SPEC 2017. Under single-core configuration, PAIC improves performance over Least Recently Used (LRU) replacement policy by 37.22%, compared with improvements of 32.93% for Signature-based Hit Predictor (SHiP), 34.56% for Hawkeye, and 34.43% for Glider. Under the four-core configuration, PAIC improves performance over LRU by 20.99%, versus 13.23% for SHiP, 17.89% for Hawkeye and 15.50% for Glider.

Keywords hardware prefetching, machine learning, Prefetch-Adaptive Intelligent Cache Replacement Policy (PAIC), replacement policy

1 Introduction

The latest research seeks to make use of machine learning for improving system performance, including branch prediction in modern microprocessors[1], cache replacement[2, 3], and memory scheduling[4]. The intelligent cache replacement policies based on machine learning algorithms[2, 3] use program counter (PC), memory address, etc., as features to learn past caching behaviors and predict future caching priorities, thus effectively improving the management of the last-level cache (LLC). Relative to the traditional heuristic-based cache replacement policies[5, 6], learning-based cache replacement policies improve the accuracy of cache line reuse prediction under complex access modes[7, 8], thereby making more accurate decisions for insertion and eviction.

Prefetching data into the cache hierarchy before actual references hides memory latency, thus significantly improving the performance. However, harmful prefetching can cause cache pollution and interfere with cache management, thus leading to performance degradation[9, 10]. In the presence of prefetching, the current learning-based cache replacement policies may provide minimal performance improvements or degrade the performance. Most of replacement policies do not distinguish between prefetch and demand requests. Thus, these replacement policies are unable to distinguish useful prefetches from useless prefetches in most cases.

Most existing work on prefetch-aware cache replacement focuses on minimizing the cache pollution due to inaccurate prefetchers. Several solutions[9, 11] involve fine-tuned cache insertion and replacement

Regular Paper

The work was supported by the Natural Science Foundation of Beijing under Grant No. 4192007 and the National Natural Science Foundation of China under Grant No. 61202076.

*Corresponding Author

priorities for prefetches to mitigate the degree of prefetch-induced LLC pollution. In this paper, we propose Prefetch-Adaptive Intelligent Cache Replacement Policy (PAIC). PAIC also deals with inaccurate prefetches. It learns to discard inaccurate and redundant prefetches at the PC granularity. By assigning low priority to both demand loads and prefetches that are likely to be prefetched again, PAIC can make better use of the cache, thus reducing prefetch-induced cache pollution.

The main goals of PAIC are 1) to improve the accuracy of reuse prediction for both prefetch and demand requests and 2) to avoid cache pollution due to harmful prefetching. PAIC accomplishes both these goals using different predictors based on Integer Support Vector Machine (ISVM)[3] for prefetch and demand requests. Thus, PAIC reduces cache pollution due to prefetch requests and combines the performance benefits from hardware prefetching and intelligent cache replacement.

We show that the intelligent cache replacement policy that adapts to prefetching improves the re-reference predictions for prefetch and demand requests. In general, this paper makes the following contributions.

• We characterize the state-of-the-art learning-based cache replacement policies in the presence and absence of hardware prefetching. Learning-based cache replacement policies improve application performance, and prefetching further improves performance by extracting useful data into the cache in advance. However, we show that without a prefetch-adaptive replacement policy, prefetching will interfere with the management of the LLC, thereby causing performance degradation and variability.

• We propose PAIC, whose novelty is two-folded. First, PAIC applies machine learning to design different cache replacement predictors for prefetch and demand requests. Second, it improves the accuracy of reuse predictions for both prefetch and demand requests, which combines the performance benefits from prefetching and cache replacement.

• Finally, we evaluate PAIC in detail with SPEC 2006 and the latest SPEC 2017 benchmarks. On average, PAIC provides a performance improvement of 36.61% and 21.46% for single-core memory-intensive applications and the four-core configuration, respectively. PAIC can adapt to different data prefetchers, and bring more effective performance improvement relative to the state-of-the-art replacement policies.

The remainder of this paper is organized as follows. In Section 2, we discuss three fundamental observations that motivate us to propose PAIC. Section 3 describes implementation details of PAIC. We then describe our solution and empirically evaluate it in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

2 Characterization & Analysis of Replacement Policies

To explore the impact of hardware prefetching on the performance of cache management, we model the signature path prefetcher (SPP)[12] to evaluate three representative learning-based cache replacement policies in the presence and absence of prefetching: Signature-based Hit Predictor (SHiP)[7], Hawkeye[8], and Glider[3], whose performance will be evaluated in the presence of hardware prefetching. SHiP is the first to use signatures (e.g., the miss-causing PC, the instruction sequence leading to the load) to predict the re-reference interval of an incoming cache line. Hawkeye is the winner of the 2nd JILP Cache Replacement Championship[①]. Glider is the first to apply deep learning to predict reuse distance.

2.1 Prefetch and Demand Requests

From the cache management perspective, the cache access patterns including prefetch and demand requests exhibit different properties. A demand request is from the processor for instructions or data that are known to be required by the processor. A prefetch request is issued by the prefetcher ahead of the processor to hide the entire latency of memory accesses. However, prefetched blocks can potentially pollute the cache by evicting the blocks which are more useful. In general, cache lines inserted into the LLC by demand requests are more likely to be performance-critical than those by prefetch requests. Moreover, prefetching does not always improve performance and sometimes may even degrade it. For example, useless prefetch requests unnecessarily consume valuable off-chip bandwidth, and useless prefetched data may evict a useful block from the cache.

[①]The 2nd cache replacement championship. http://crc2.ece.tamu.edu/, May 2021.

Many replacement policies do not distinguish between prefetch and demand requests. Such policies are more likely to cause cache pollution and increase the number of cache misses.

2.2 Performance Variability Under Various Replacement Policies with Prefetching

Fig.1 shows the application performance with the aforementioned cache replacement policies in the presence of prefetching. The graph shows that prefetching can significantly improve application performance by approximately 32%. However, for the performance of individual applications in each workload category, we observe significant variations across different cache replacement policies. Fig.2 and Fig.3 show the performance variabilities in the absence and presence of prefetching, respectively. In both figures, the x-axis represents different workloads in the experiment, and the y-axis represents the performance relative to the baseline Least Recently Used (LRU) replacement policy[13] without prefetching.

Fig.2 shows that in the absence of prefetching, the learning-based cache replacement policies improve the performance of various applications. However, in the presence of prefetching, we observe differential performance effects. For example, sphinx3 achieves 19%–22% performance improvement over LRU from learning-based cache replacement policies in the absence of prefetching. However, as shown in Fig.3, in the presence of prefetching, the improvement of sphinx3 over LRU reduces to 0.22% on average.

Fig.3 illustrates that in the presence of prefetching, the performance improvements from learning-based cache replacement policies degrade significantly. Designing an intelligent cache replacement policy that adapts to prefetching will further improve the performance of such applications in the presence of prefetching.

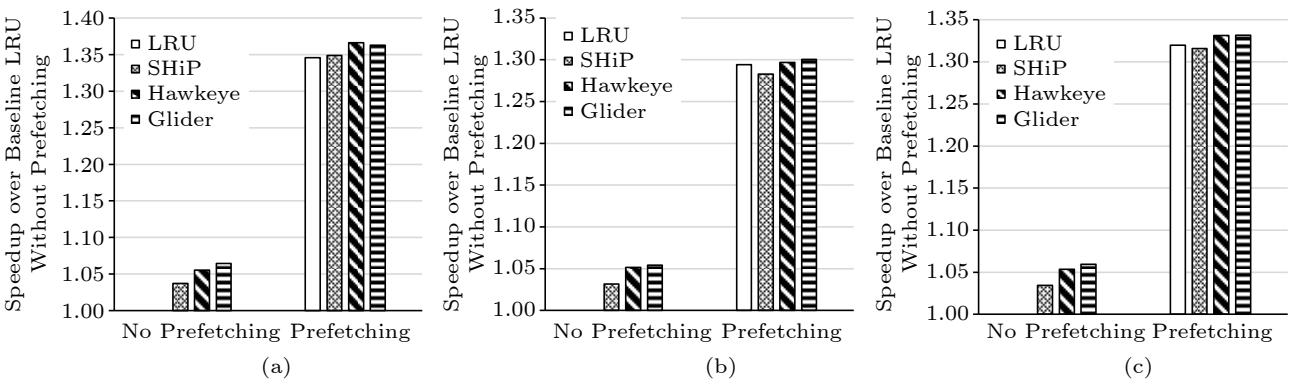

Fig.1. Hardware prefetching significantly improves application performance. (a) Speedup for SPEC 2006. (b) Speedup for SPEC 2017. (c) Speedup for all benchmarks.

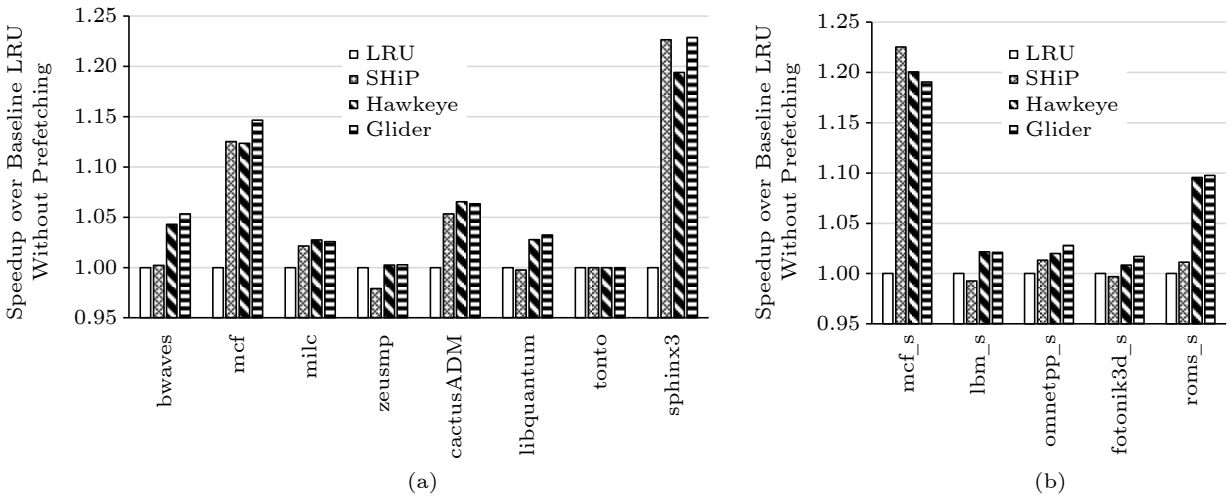

Fig.2. Performance of individual applications under cache replacement policies without prefetching. (a) Speedup for SPEC 2006. (b) Speedup for SPEC 2017.

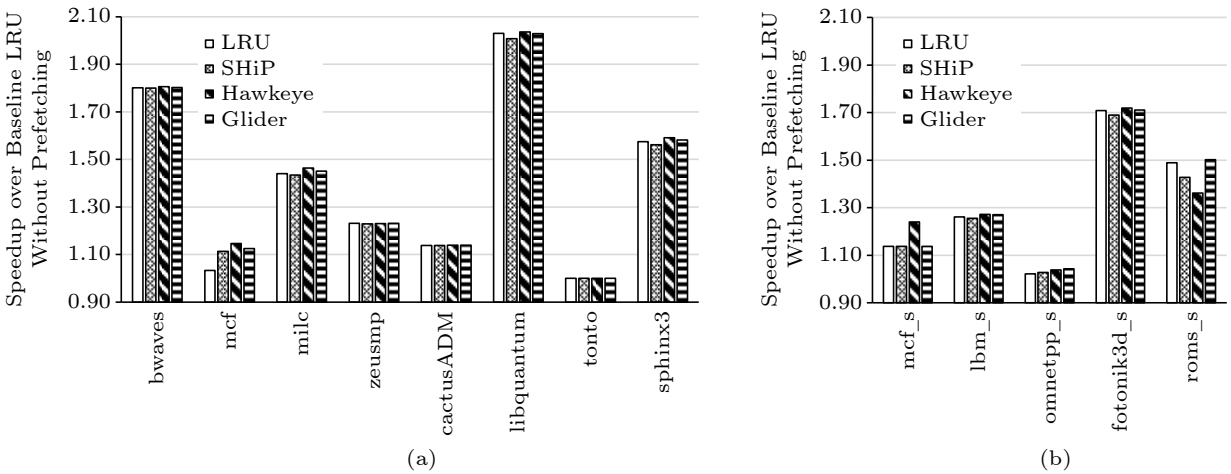

Fig.3. Performance of individual applications under cache replacement policies in the presence of prefetching. (a) Speedup for SPEC 2006. (b) Speedup for SPEC 2017.

2.3 Low LLC Reuse of Prefetch Requests

Previous studies[10, 11, 14] have shown that a large fraction of prefetches are dead in the LLC. Low reuse of prefetched cache lines at the LLC can be attributed to either prefetcher pollution or timely data prefetching by upper levels of the cache hierarchy. Subsequent demand requests are directly served in L2 and never reach the LLC. As a result, the filtering of temporal locality by smaller caches leads to lower reuse of prefetched lines in the LLC. Therefore, we propose a replacement policy that can predict the re-reference of prefetch requests, and preferentially evict useless prefetches, thus improving the performance of applications under prefetching.

In summary, the appearance of cache-averse prefetched cache lines at the LLC and reduced gains from learning-based cache replacement policies in the presence of hardware prefetching necessitate the need for PAIC. We describe PAIC in detail in Section 3.

3 PAIC Design and Implementation

Prefetch requests have different properties relative to demand requests. Generally, cache lines inserted into LLC by demand requests are more likely critical for the performance relative to the prefetch requests. PAIC distinguishes the caching behaviors of demands and prefetches through different predictors for the corresponding requests.

To mitigate the degree of prefetch-induced interference, PAIC makes a re-reference prediction at the granularity of a request type. Specifically, based on a rich dynamic program context, two hardware-friendly ISVM models are designed, each of which uses a memory access sequence, thus significantly improving the prediction of re-reference for prefetch and demand requests. Thus, PAIC effectively caches the most useful demand- and prefetch-requested data and distinguishes useful prefetches from useless prefetches.

Fig.4 shows the overall structure of PAIC. PAIC can reduce cache pollution by distinguishing between demand loads and prefetches. PAIC uses two PC-based predictors to determine whether a demand load or a prefetch is likely to be prefetched again and uses this information to insert such lines with a low priority. Such low priority lines will be preferentially evicted and can mitigate prefetch-induced interference.

PAIC's main components comprise the PAIC predictor which makes eviction decisions and DMINgen which simulates the Demand-MIN[15] to produce labels that train the PAIC predictor. Each component will be described in more detail.

3.1 DMINgen

DMINgen determines the prospective cache if Demand-MIN[15] has been used. Demand-MIN evicts the line that is prefetched furthest in the future, thus minimizing the number of demand misses. Fig.5 shows an access sequence, wherein demand loads are shown in blue and prefetches in yellow. For a cache that can hold two cache lines and initially holds A and B, when line C is loaded into the full cache, the eviction of A or B will result in different numbers of demand misses.

Belady's MIN algorithm[8] produces two demand misses. The first demand miss occurs at $t = 1$, when

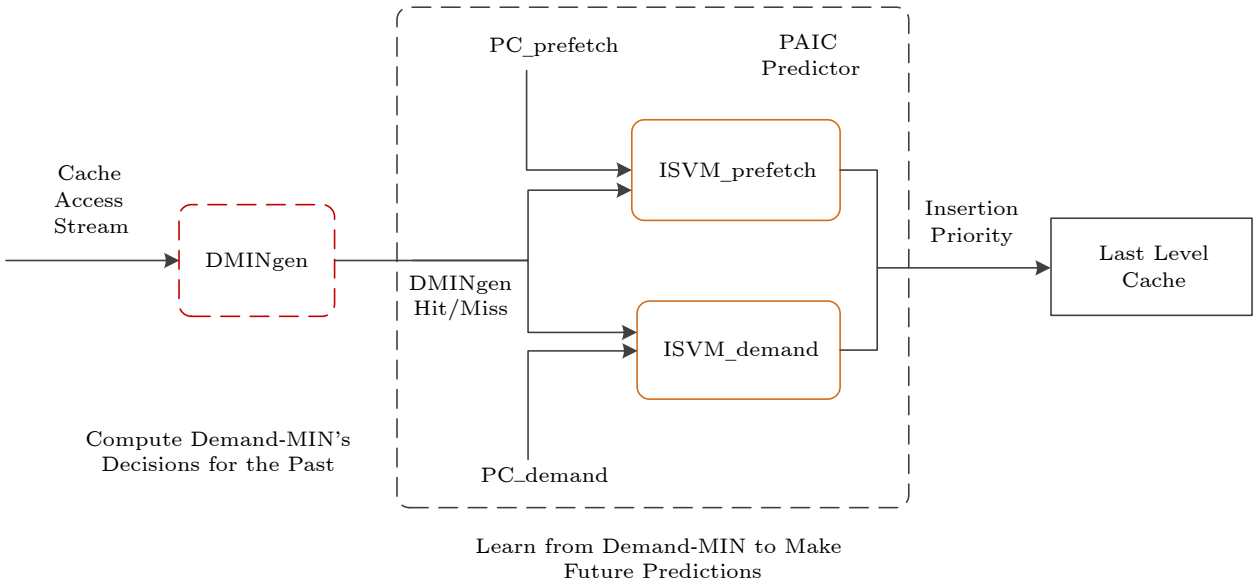

Fig.4. Overview of PAIC.

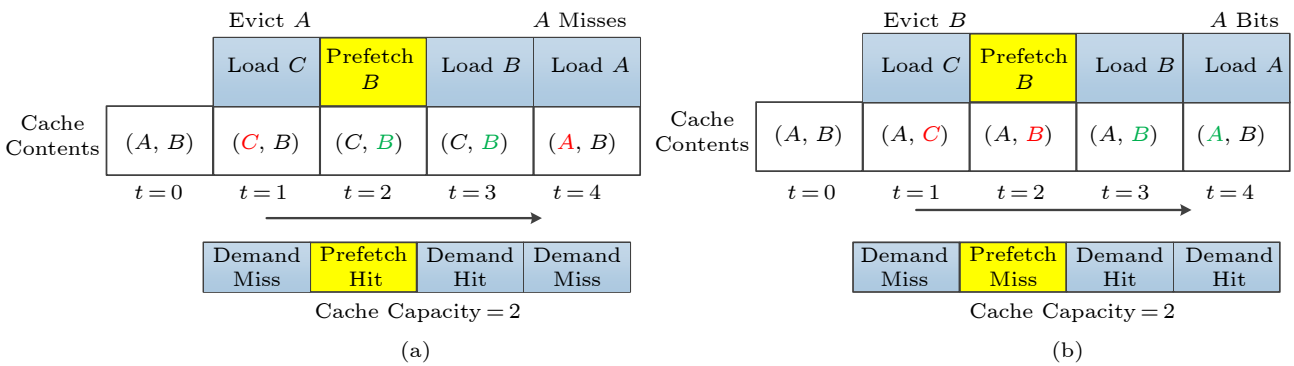

Fig.5. (a) Belady's MIN results in two demand misses. (b) Demand-MIN results in one demand miss.

line C is loaded into a full cache. Fig.5(a) shows that Belady's MIN algorithm evicts A, which is reused further in the future than B. The second demand miss occurs at $t = 4$, when A is loaded.

B will be prefetched at $t = 2$, such that the demand reference to B at $t = 3$ hits irrespective of our decision at $t = 1$, thus reducing the number of demand misses. Fig.5(b) shows at $t = 1$, B gets evicted instead of A, and the demand reference to B at $t = 3$ and that to A at $t = 4$ both hit in the cache. Thus, the Demand-MIN exchanges a prefetch hit for a demand hit, resulting in a single demand miss (to C). Here, to minimize demand misses, Demand-MIN preferentially evicts B, which is prefetched furthest in the future.

To reconstruct the Demand-MIN algorithm for past accesses, we extend the concept of usage interval defined in Hawkeye (the interval between two consecutive references to the same cache line). To distinguish between demand loads and prefetches, we identify the endpoint of the usage interval as being a demand access (D) or a prefetch (P). Four types of usage intervals are obtained: D-D, P-D, P-P, and D-P. If we include open intervals, representing lines that are never reused, there would be two more types: P-open and D-open. The first two types of intervals (D-D and P-D) are cached by Demand-MIN if there is space in the cache, while the last two types (D-open and P-open) are preferentially evicted by Demand-MIN as they will never be reused.

The benefit of Demand-MIN lies in evicting intervals that end with a prefetch. For the intervals that end with a prefetch, including D-P and P-P, demand hits are not yielded; therefore Demand-MIN evicts these intervals to make room for other intervals to yield demand hits.

3.2 PAIC Predictor

The second major component of PAIC comprises two different sub-predictors (the prefetch sub-predictor and the demand sub-predictor) based on ISVM. We separate the prediction for prefetch and demand requests using the two sub-predictors. The caching behaviors of load instructions resulting in prefetch and demand accesses may be different and cannot be treated equally. For example, a load instruction that loads cache-friendly demand accesses but issues inaccurate prefetches will be classified as cache-friendly by the demand predictor and cache-averse by the prefetch predictor.

PAIC uses a PC-based prefetch sub-predictor to provide replacement priorities for prefetches. For example, the prefetch sub-predictor can learn that prefetches triggered by a certain PC are more likely inaccurate relative to those triggered by a different PC. PAIC mitigates the degree of prefetch-induced cache interference by evicting inaccurate prefetches.

PAIC improves the prediction accuracy by exploiting richer dynamic program context, i.e., the most recent PCs of memory access instructions that accessed LLC. Previous work[3] has shown that a longer history of past PCs would benefit predictive replacement policies in the LLC, and the prediction accuracy is insensitive to the order of these PCs but depends on the existence of important PCs. Therefore, we remove the repeated PCs in the memory access sequence, ignore the order between PCs, and feed these simplified features into ISVM.

Fig.6 shows the hardware implementation of the PAIC predictor, which is comprised of three main components: 1) a PC history register (PCHR), 2) a prefetch ISVM table (the ISVM_pf table) corresponding to the prefetch sub-predictor, and 3) a demand ISVM table (the ISVM_de table) corresponding to the demand sub-predictor. The PCHR maintains an unordered list of the last k PCs for each core. We model PCHR as a small LRU cache that tracks the last k unique PCs (in our experiments, we set $k = 5$). There is an ISVM for each PC, and the ISVM table tracks the weights of each PC's ISVM. To distinguish re-reference predictions for prefetch and demand requests, separate ISVM tables for prefetch and demand accesses are used. We model each table as a direct-mapped cache, indexed by the hash of the current PC (PC_{current}), thus returning its corresponding ISVM weights.

Each PC's ISVM consists of 16 weights for different possible PCs in the history register. To identify the weight corresponding to each PC in the PCHR, a 4-bit hash is created for each element in the PCHR, and we retrieve these weights in the prefetch or demand ISVM table. For example, as shown in Fig.6, as the current prefetch access, PCHR contains PC 1, PC

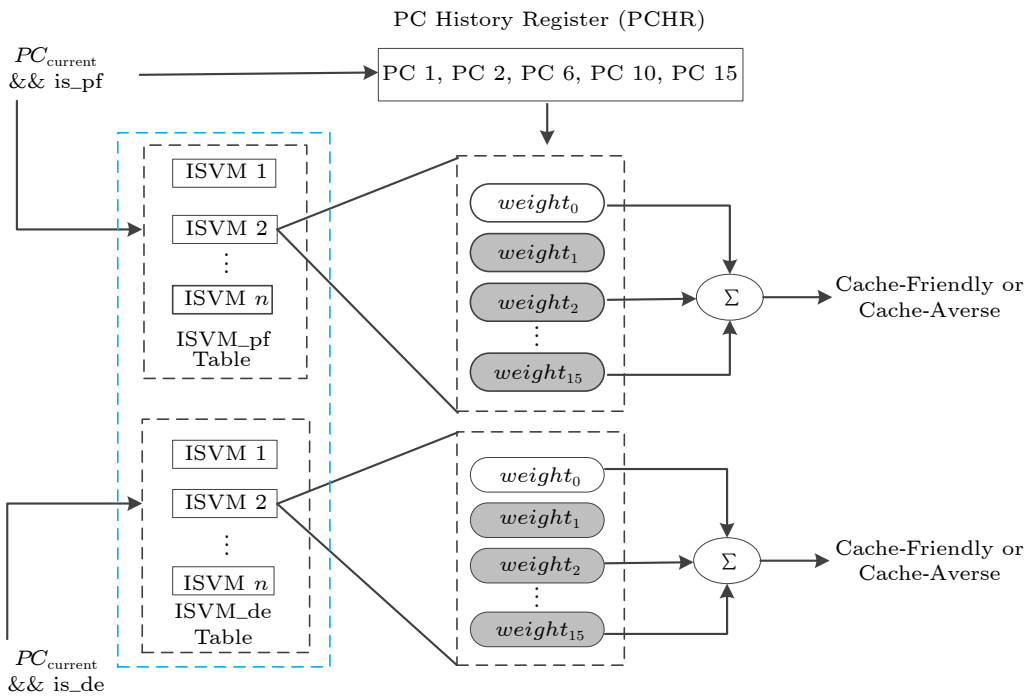

Fig.6. PAIC predictor.

2, PC 6, PC 10, and PC 15, and we retrieve $weight_1$, $weight_2$, $weight_6$, $weight_{10}$, and $weight_{15}$ in the ISVM_pf table for both training and prediction.

Below we discuss the training and prediction operations of the PAIC predictor in more detail.

3.2.1 Training

PAIC's predictors are trained based on the behaviors of a few sampled sets[8]. Upon access to a sampled set, according to the type of requests (i.e., demand or prefetch), PAIC retrieves the weights corresponding to the current PC and the PCHR. If DMINgen determines that the cache line should be cached, the weights will be increased by 1; otherwise they will be decreased by 1. The weights are not updated if their sum is above a certain threshold. To set a reasonable threshold, PAIC dynamically selects a threshold from a set of fixed ones. PAIC can better learn the caching behaviors of load instructions that result in prefetch and demand accesses by distinguishing between the prefetch and demand requests.

3.2.2 Prediction

PAIC improves the prediction accuracy in the presence of prefetching with two different predictors to separate the prediction of prefetch and demand requests. To make predictions at the granularity of a request type, the weights corresponding to the current PC and the PCHR are summed. If the summation is greater than or equal to the threshold, we predict that the line is cache-friendly and insert it with high priority (RRPV = 0). If the summation is less than 0, we predict that the line is cache-averse and insert it with low priority (RRPV = 7). For the remaining cases, we determine that the line is cache-friendly with a low confidence and insert it with medium priority (RRPV = 2). The re-reference prediction value (RRPV)[6] indicates the relative importance of cache lines.

3.3 Hardware Overhead

For a 2 MB LLC, PAIC's budgets for replacement state per line, sampler, and DMINgen are 12 KB, 12.7 KB, and 4 KB, respectively. The overhead of PAIC mainly comes from the ISVM-based PAIC predictor. For each ISVM, we track 16 weights, and each weight is 8-bit wide. Therefore, each ISVM consumes 16 bytes. Since we track 2 048 PCs and distinguish between prefetch and demand requests, PAIC's predictors consume a total of 65.5 KB. The PCHR with a history of the last five accesses is only 0.1 KB. Therefore, the total hardware budget of PAIC is 94.3 KB. Since the PAIC's predictors require only two table lookups for both training and prediction, these predictors' latency can be easily hidden by the latency of accessing the LLC.

Designing separate predictors for prefetch and demand requests makes a significant impact on performance as shown in Section 4. Given the results of the ISVM table size tuning, the hardware overhead of PAIC is saved by reducing the entries of ISVM tables as described in Subsection 4.2.

4 Evaluation

4.1 Methodology

We evaluate PAIC using the simulation framework released by the 2nd JILP Cache Replacement Championship (CRC2). The framework is based on ChampSim and models a three-level cache hierarchy.

Hardware Prefetcher. To prove that PAIC can adapt to different hardware prefetchers, we model a signature path prefetcher (SPP)[12] and a Kill-the-PC prefetcher (KPC-P)[16]. SPP and KPC-P are the latest examples of forward-looking prefetchers, and both use prefetch filters trained using the L2 cache access (hits and misses) as feedback.

Benchmarks. We evaluate PAIC on the 27 memory-intensive applications of SPEC CPU2006[17] and SPEC CPU2017[18] benchmark suites. We run the benchmarks using the reference input set, and as with the CRC2, SimPoint[19] is used to generate a single sample of one billion instructions per benchmark. We warm the cache for 200 million instructions and evaluate the behavior of the next one billion instructions.

Multi-Core Workloads. Our multi-core results simulate four benchmarks running on four cores, choosing 60 mixes from all possible workload mixes. For each mix, we simulate the simultaneous execution of the SimPoint samples of the constituent benchmarks until at least 250 million instructions are executed per benchmark.

To evaluate performance, we report the weighted speedup normalized to LRU for each benchmark mix. The metric is computed as follows: for each program sharing cache, we compute its instructions per cycle (IPC) in a shared environment (IPC_{shared}), and in iso-

lation on the same cache (IPC_{single}). We compute the weighted IPC of a mix as the sum of IPC_{shared} /IPC_{single} for all benchmarks in the mix and normalize it with the weighted IPC using the LRU replacement policy.

Baseline Replacement Policies. We compare PAIC against five state-of-the-art cache replacement policies, namely, SRRIP, DRRIP[6], SHiP[7], Hawkeye[8], and Glider[3]. SRRIP and DRRIP are variations of the LRU policy, which use the Re-reference Interval Prediction (RRIP), aiming at preventing blocks with a distant re-reference interval from cache pollution. SHiP extends RRIP by predicting the re-reference interval of an incoming cache line based on its history. Hawkeye has won the 2017 Cache Replacement Championship[8]. Glider is the first to apply deep learning to predict reuse distance[3].

4.2 Comparison with Other Policies

4.2.1 *Single-Core Performance*

We compare PAIC with five state-of-the-art replacement policies: SRRIP, DRRIP, SHiP, Hawkeye, and Glider. Fig.7 shows that relative to the five replacement policies, PAIC effectively integrates the benefits of hardware prefetching and replacement policies with SPP. Fig.7(a) and Fig.7(b) show the performance improvement of each replacement policy for

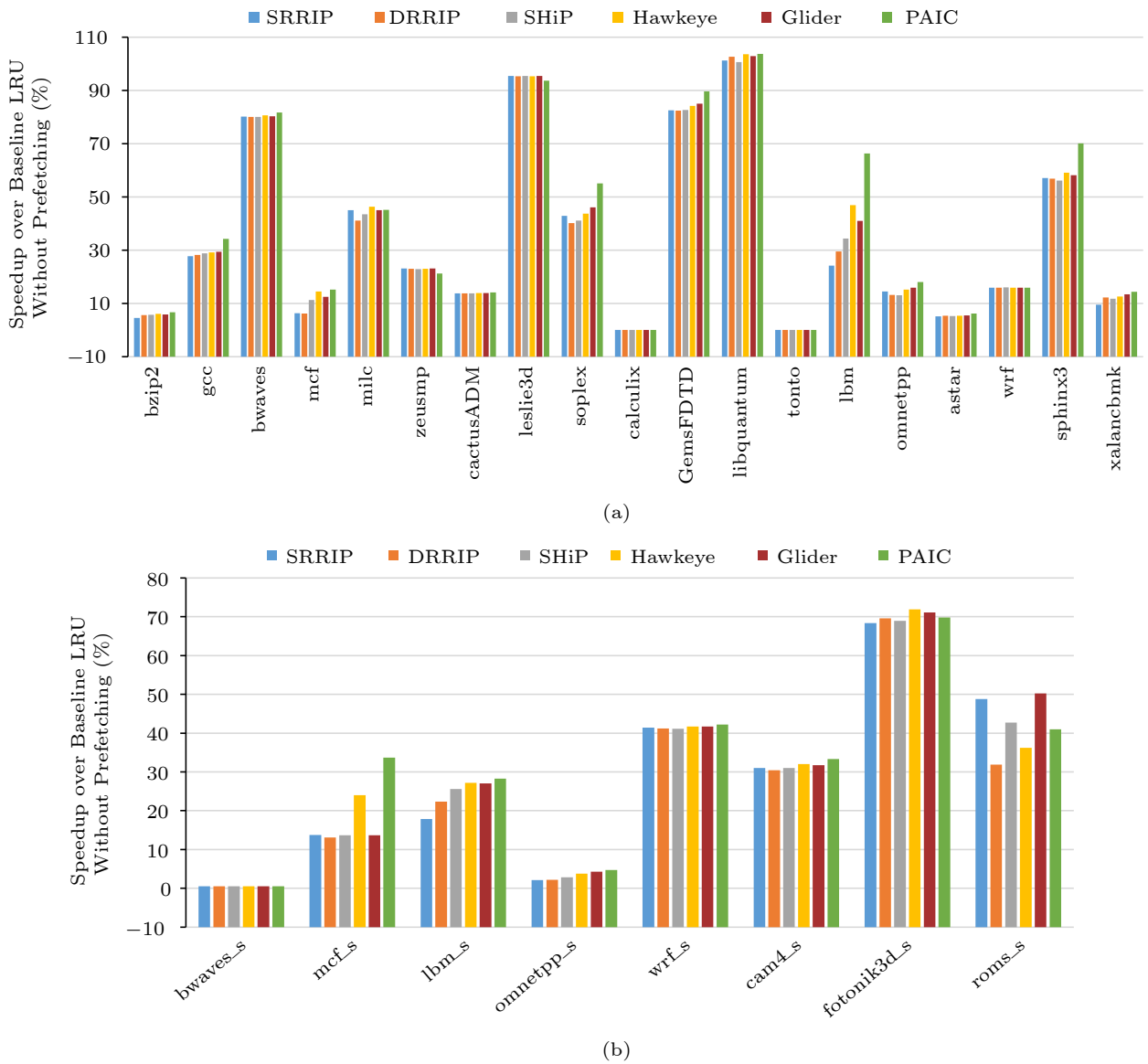

Fig.7. Speedup for single-core benchmarks. (a) Speedup for SPEC 2006 single-core benchmarks. (b) Speedup for SPEC 2017 single-core benchmarks.

SPEC 2006 and SPEC 2017, respectively. On average, PAIC improves the performance over the baseline (LRU without prefetching) by 37.22%. SRRIP, DRRIP, SHiP, Hawkeye, and Glider, in contrast, improve performance over LRU by 32.35%, 31.96%, 32.93%, 34.56%, and 34.43%, respectively. These improvements indicate that PAIC can improve the performance significantly by distinguishing between prefetch and demand requests.

We also compare PAIC, SRRIP, DRRIP, SHiP, Hawkeye, and Glider with the baseline, which is LRU with prefetching. In the presence of SPP, PAIC improves the performance by 3.86% compared with the baseline, while SRRIP, DRRIP, SHiP, Hawkeye, and Glider improve the performance by 0.04%, −0.18%, 0.58%, 1.85%, and 1.68%, respectively. On average, in the presence of prefetching, learning-based cache replacement policies provide minimal performance improvements or unexpectedly degrade performance. PAIC can address some of the performance degradation caused by prefetcher pollution.

4.2.2 *Multi-Core Performance*

Fig.8 shows that in the presence of hardware prefetching, PAIC can improve performance of a four-core system. With SPP, PAIC achieves a speedup of 20.99%, while SRRIP, DRRIP, SHiP, Hawkeye, and Glider achieve speedups of 13.83%, 12.74%, 13.23%, 17.89%, and 15.50%, respectively. PAIC distinguishes between prefetch and demand requests, and exploits the dynamic program context to improve the prediction accuracy. Overall, PAIC demonstrates an effective intelligent replacement policy by differentiating the predictions for prefetch and demand requests.

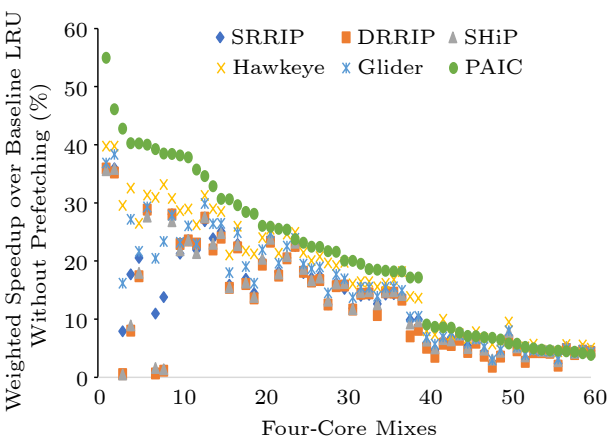

Fig.8. Weighted speedup for four cores with a shared 8 MB LLC.

4.2.3 *Effect of PAIC with KPC-P*

We also evaluate PAIC with KPC-P[16] and find that the results are similar to those achieved with SPP. On average, with KPC-P, PAIC improves the performance over the baseline by 36.61% in a single-core setting, while SRRIP, DRRIP, SHiP, Hawkeye, and Glider improve performance over the baseline by 31.08%, 31.52%, 32.25%, 33.64%, and 33.63%, respectively.

Fig.9(a) and Fig.9(b) show the LLC demand miss reduction of each replacement policy for SPEC 2006 and SPEC 2017, respectively. On average, Fig.9 shows that in the presence of KPC-P, PAIC achieves an average demand miss reduction of 17.41%, while SRRIP, DRRIP, SHiP, Hawkeye and Glider achieve demand miss reductions of 12.81%, 13.30%, 13.66%, 15.46%, and 15.48%, respectively. These improvements indicate that PAIC can support different prefetchers and improve the performance significantly.

4.3 Performance Overview

4.3.1 *Evaluation of Prefetch Filter*

The above experimental results show that, with different data prefetchers, PAIC performs better than the state-of-the-art cache replacement policies. Perceptron-based Prefetch Filtering (PPF)[20] enables more aggressive tuning of the SPP underlying prefetcher, leading to increased coverage by filtering out the inaccurate prefetches. We compare PAIC+SPP+PPF with PAIC+SPP to further evaluate PAIC's utility in the presence of a prefetch filter with SPP as the underlying prefetcher. As shown in Fig.10, PAIC+SPP improves performance over the baseline by 37.22%, while PAIC+SPP+PPF yields a speedup of 44.31% over the baseline. It is evident that PAIC consistently achieves high performance under the influence of prefetching with or without a prefetch filter.

4.3.2 *Effective Sequence Length*

Fig.11 shows the relationship between history length and speedup of PAIC, wherein the number of unique PCs (the k value described in Subsection 3.2) for PAIC ranges from 3 to 7. We make two observations. First, PAIC is insensitive to finer changes in the k value; for SPP and KPC-P, PAIC achieves the best performance with four and six unique PCs, respectively. Second, ISVM-based sub-predictors of PA-

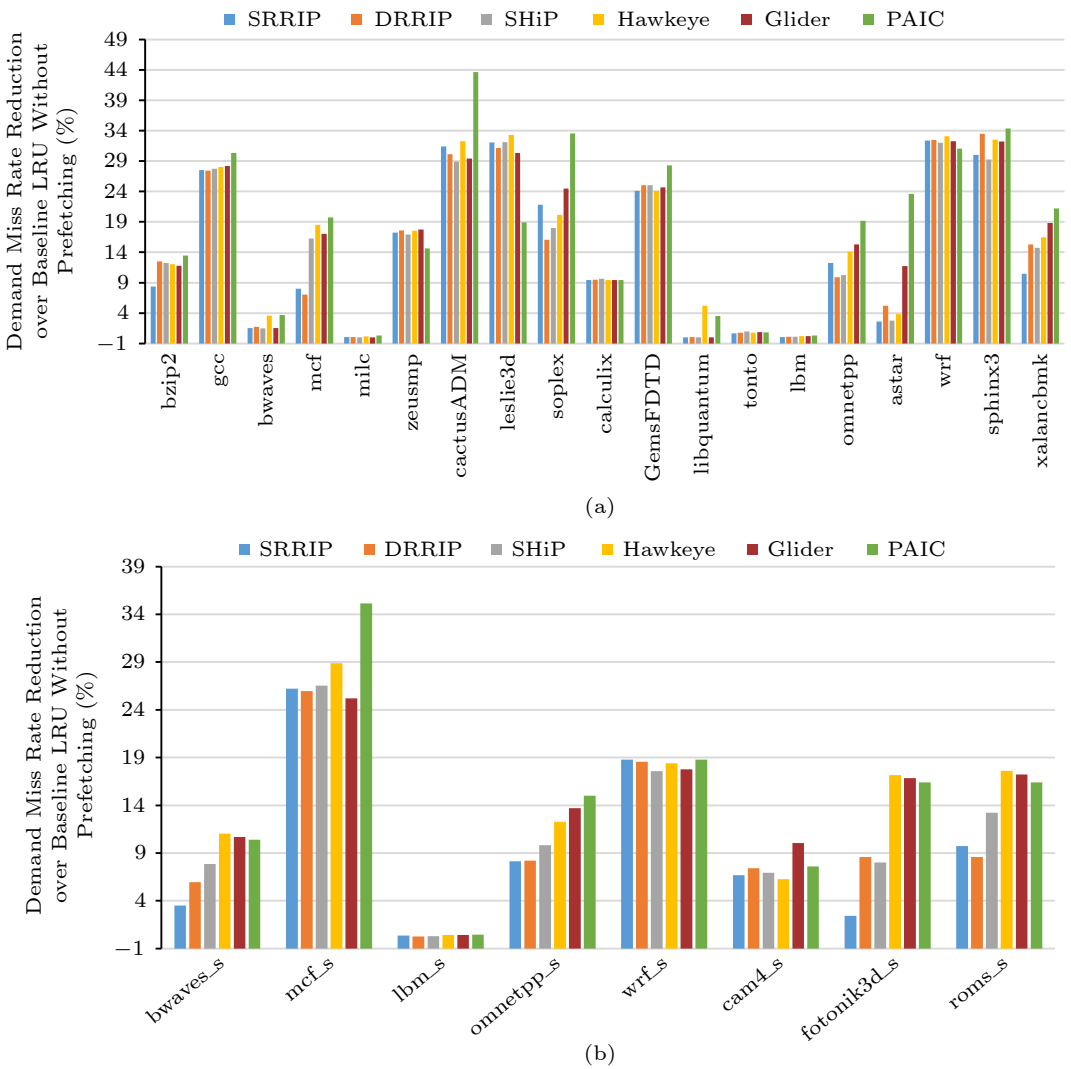

Fig.9. Demand miss rate reduction for single-core benchmarks. (a) Demand miss rate reduction for SPEC 2006 single-core benchmarks. (b) Demand miss rate reduction for SPEC 2017 single-core benchmarks.

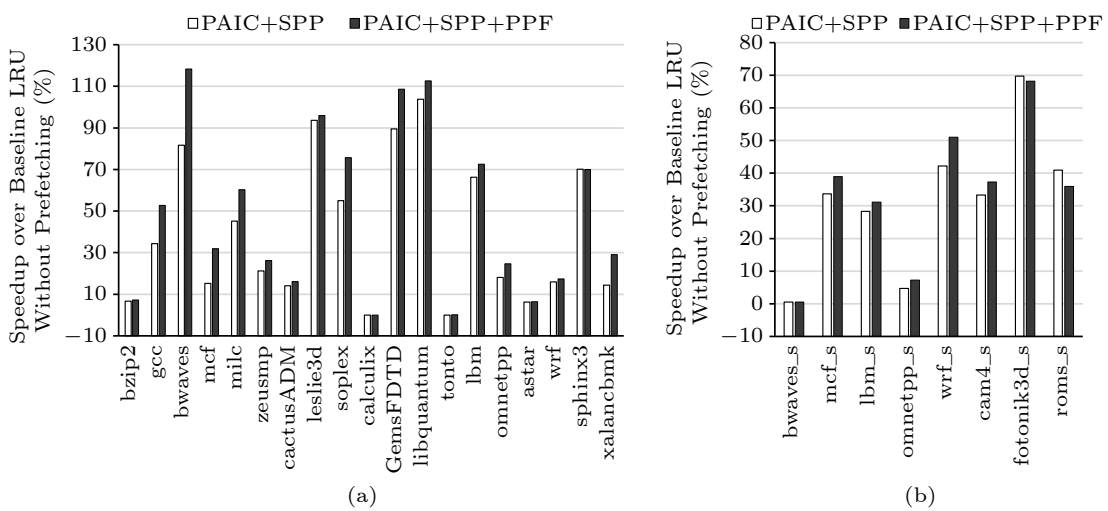

Fig.10. Speedup for single-core benchmarks. (a) Speedup for SPEC 2006. (b) Speedup for SPEC 2017.

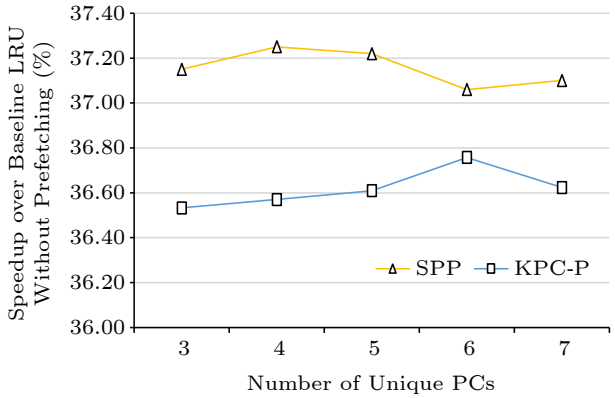

Fig.11. Performance of PAIC under different numbers of unique PCs.

IC effectively identify the important memory accesses with fewer history elements. PAIC makes optimal replacement decisions depending on the presence of the few important PCs and uses separate sub-predictors for prefetch and demand requests. Thus, PAIC significantly improves the prediction of caching behaviors in the presence of prefetching through a simple hardware-friendly linear model.

4.3.3 *Reducing the Hardware Overhead*

We take tests to check the impact of reducing ISVM table sizes on PAIC's performance. As shown in Fig.12, the performance degradation is marginal by significantly reducing the perceptron table sizes for more space-saving. Losing only 0.07% performance by reducing the table sizes to only 16 entries is acceptable in adapting a predictor to a smaller ISVM table. Each ISVM table is reduced to 16 entries, resulting in a total consumption of 0.51 KB from PAIC's predictors. Therefore, the total hardware budget of PAIC is reduced to 29.31 KB (reduced to about 1/3 of the original).

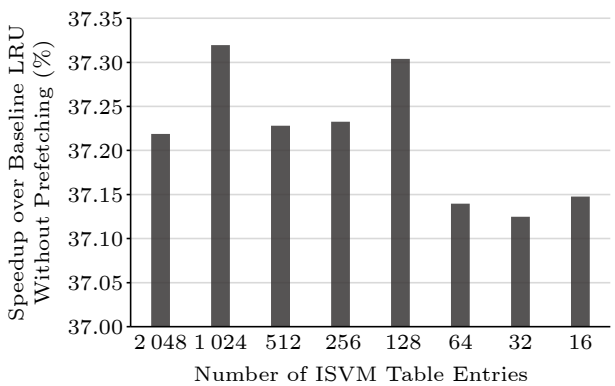

Fig.12. Performance of PAIC under different numbers of ISVM table entries.

Table 1 shows the comparison of hardware overhead and performance between several prior state-of-the-art policies and PAIC. PAIC uses little hardware beyond Hawkeye; however, on average, it offers 2.90% additional performance improvement for the SPEC 2006 applications and 1.85% for the SPEC 2017 applications. Overall, PAIC offers a practical machine learning based cache design for eliminating prefetch-induced interference in the shared LLC.

Table 1. Hardware Overhead and Performance Comparison for Different Replacement Policies

Policy	Speedup over LRU (%)		Hardware
	SPEC 2006	SPEC 2017	Overhead (KB)
SRRIP[6]	34.18	27.99	8.00
DRRIP[6]	34.30	26.41	8.00
SHiP[7]	34.87	28.30	14.00
Hawkeye[8]	36.62	29.67	28.00
Glider[3]	36.28	30.04	61.60
PAIC	39.52	31.52	29.31

5 Related Work

5.1 Hardware Prefetchers

Prefetching improves performance by preloading cache lines into some level of the cache before actually being used. Typically, prefetchers predict future access patterns based on past memory accesses. Shevgoor *et al.*[21] proposed a variable-length incremental prefetcher to identify complex address patterns repeated in physical pages. SPP[12] learns and prefetches complex data access patterns using compressed historical signatures. It achieves high accuracy by correlating the signature with future possible delta patterns. KPC-P[16] aims to generate prefetch confidence values, which control the aggressiveness of prefetching. A prefetch request with low confidence indicates its long use distance, or it is an inaccurate prefetch request. KPC-P avoids L2 cache pollution by no longer inserting of prefetched lines with low prediction confidence in L2. However, all prefetched lines are inserted in LLC. PAIC avoids prefetch-induced cache pollution in LLC by evicting cache lines that are prefetched furthest in the future, and freeing up cache space for cache lines of other access types.

5.2 Prefetch Filters

The prefetch filter is implemented through a standalone module that examines the addresses generated from the prefetcher. The prefetch filter enhances an underlying prefetcher by filtering out predicted un-

used prefetches. To further improve the effectiveness of hardware prefetchers, prefetch filters like PPF[20] and Evicted-Prefetch-Filter (EPF)[22] have been proposed. PPF is an intelligent prefetch filter that enhances the SPP prefetcher. PPF filters out useless prefetching issued by aggressive SPP, thus allowing aggressive predictions without degrading the accuracy.

5.3 Learning-Based Cache Replacement Policies

Cache replacement policies essentially predict the re-reference behaviors of cache lines[6]. The state-of-the-art replacement policies[2, 3, 7, 8] are based on some signatures, including PCs, memory addresses, or instruction sequences that learn from past cache behaviors to predict future cache priorities. The goal of these policies is to predict whether an incoming line is cache-friendly or cache-averse. For example, SHiP[7] monitors evictions from a sampler to learn the re-reference interval of a given load instruction. Hawkeye[8] learns from the Belady-MIN algorithm based on past cache accesses and maintains a table of counters to learn whether memory accesses by a given PC are cache-friendly or cache-averse.

Only a few previous studies have applied machine learning solutions to the cache replacement problem, and these do not distinguish between prefetch and demand requests. A recent study[2] uses online perceptron learning to improve the accuracy of cache reuse prediction. It uses a short and ordered history of PCs as the input feature to predict the reuse of cache blocks. Shi *et al.*[3] applied an attention-based long short-term memory (LSTM) for training reuse predictors offline and extracted useful insights from historical PCs. These insights are then used to build an online SVM-based hardware predictor, namely, the Glider replacement policy. However, in the presence of prefetching, Glider makes the same predictions for prefetch and demand requests, degrades prediction accuracy, and provides minimal performance improvements, as it cannot avoid prefetch-induced cache pollution. Herein, we propose that PAIC can improve the accuracy of cache line reuse prediction in the presence of prefetching, along with the performance of the memory subsystem.

6 Conclusions

In this paper, to solve the problem of performance degradation and variability under prefetching, we proposed an intelligent cache replacement policy (PAIC) in the presence of prefetches. PAIC uses different sub-predictors for prefetch and demand requests and inputs a compressed long history of past PCs into ISVM-based sub-predictors, which significantly improves reuse prediction accuracy for both prefetch and demand requests. By distinguishing the caching behaviors of prefetches from demands, PAIC achieves better learning from past memory accesses at the granularity of a request type. In the single-core configuration, PAIC improves performance over the baseline (LRU without prefetching) by 37.22%, outperforming the state-of-the-art replacement policy (Glider) by 2.79%. On a four-core system, PAIC improves performance over the baseline by 20.99%, beating Glider by 5.49%. Overall, PAIC offers a guide to applying machine learning to combine the performance benefits from hardware prefetching and cache replacement policies.

In the context of a multi-core processor, cache coherence is a long-standing question that has largely been unaddressed concerning prefetching. In our future work, we will focus on the sharing issue and prefetching.

References

[1] Zangeneh S, Pruett S, Lym S, Patt Y N. BranchNet: A convolutional neural network to predict hard-to-predict branches. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.118–130. DOI: 10.1109/MICRO50266.2020.00022.

[2] Teran E, Wang Z, Jiménez D A. Perceptron learning for reuse prediction. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016. DOI: 10.1109/MICRO.2016.7783705.

[3] Shi Z, Huang X R, Jain A, Lin C. Applying deep learning to the cache replacement problem. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.413–425. DOI: 10.1145/3352460.3358319.

[4] Ipek E, Mutlu O, Martínez J F, Caruana R. Self-optimizing memory controllers: A reinforcement learning approach. *ACM SIGARCH Computer Architecture News*, 2008, 36(3): 39–50. DOI: 10.1145/1394608.1382172.

[5] Hallnor E G, Reinhardt S K. A fully associative software-managed cache design. In *Proc. the 27th Annual International Symposium on Computer Architecture*, Jun. 2000, pp.107–116. DOI: 10.1145/339647.339660.

[6] Jaleel A, Theobald K B, Steely Jr S C, Emer J. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, 2010, 38(3): 60–71. DOI: 10.1145/1816038.

1815971.

[7] Wu C J, Jaleel A, Hasenplaugh W, Martonosi M. SHiP: Signature-based hit predictor for high performance caching. In *Proc. the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2011, pp.430–441. DOI: 10.1145/2155620.2155671.

[8] Jain A, Lin C. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proc. the 43rd Annual International Symposium on Computer Architecture*, Jun. 2016, pp.78–89. DOI: 10.1109/ISCA.2016.17.

[9] Wu C J, Jaleel A, Martonosi M, Steely S C, Emer J. PACMan: Prefetch-aware cache management for high performance caching. In *Proc. the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2011, pp.442–453. DOI: 10.1145/2155620.2155672.

[10] Heirman W, Bois K D, Vandriessche Y, Eyerman S, Hur I. Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors. In *Proc. the 27th International Conference on Parallel Architectures and Compilation Techniques*, Nov. 2018, Article No. 28. DOI: 10.1145/3243176.3243181.

[11] Ishii Y, Inaba M, Hiraki K. Unified memory optimizing architecture: Memory subsystem control with a unified predictor. In *Proc. the 26th ACM International Conference on Supercomputing*, Jun. 2012, pp.267–278. DOI: 10.1145/2304576.2304614.

[12] Kim J, Pugsley S H, Gratz P V, Reddy A L N, Wilkerson C, Chishti Z. Path confidence based lookahead prefetching. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016. DOI: 10.1109/MICRO.2016.7783763.

[13] O'Neil E J, O'Neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 1993, 22(2): 297–306. DOI: 10.1145/170036.170081.

[14] Srinath S, Mutlu O, Kim H, Patt Y N. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. the 13th International Symposium on High Performance Computer Architecture*, Feb. 2007, pp.63–74. DOI: 10.1109/HPCA.2007.346185.

[15] Jain A, Lin C. Rethinking Belady's algorithm to accommodate prefetching. In *Proc. the 45th Annual International Symposium on Computer Architecture*, Jun. 2018, pp.110–123. DOI: 10.1109/ISCA.2018.00020.

[16] Kim J, Teran E, Gratz P V, Jiménez D A, Pugsley S H, Wilkerson C. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices*, 2017, 52(4): 737–749. DOI: 10.1145/3093336.3037701.

[17] Henning J L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006, 34(4): 1–17. DOI: 10.1145/1186736.1186737.

[18] Bucek J, Lange K D, von Kistowski J. SPEC CPU2017: Next-generation compute benchmark. In *Proc. the 2018 ACM/SPEC International Conference on Performance Engineering*, Apr. 2018, pp.41–42. DOI: 10.1145/3185768.3185771.

[19] Perelman E, Hamerly G, Van Biesbrouck M, Sherwood T, Calder B. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 2003, 31(1): 318–319. DOI: 10.1145/885651.781076.

[20] Bhatia E, Chacon G, Pugsley S, Teran E, Gratz P V, Jiménez D A. Perceptron-based prefetch filtering. In *Proc. the 46th International Symposium on Computer Architecture*, Jun. 2019. DOI: 10.1145/3307650.3322207.

[21] Shevgoor M, Koladiya S, Balasubramonian R, Wilkerson C, Pugsley S H, Chishti Z. Efficiently prefetching complex address patterns. In *Proc. the 48th International Symposium on Microarchitecture*, Dec. 2015, pp.141–152. DOI: 10.1145/2830772.2830793.

[22] Seshadri V, Yedkar S, Xin H Y, Mutlu O, Gibbons P B, Kozuch M A, Mowry T C. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization*, 2015, 11(4): Article No. 51. DOI: 10.1145/2677956.

Hui-Jing Yang received her B.S. degree in computer science and technology from Zhoukou Normal University, Zhoukou, in 2018. She is currently a Ph.D. candidate at Beijing University of Technology, Beijing. She is a student member of CCF. Her main research direction is computer architecture.

Juan Fang received her M.S. degree in computer science and technology from Jilin University of Technology, Changchun, in 1997, and her Ph.D. degree in computer science and technology from the College of Computer Science, Beijing University of Technology, Beijing, in 2005. In 1997, she joined the College of Computer Science, Beijing University of Technology, Beijing. Since 2015, she has been a professor of Beijing University of Technology, Beijing. Her research interests include high-performance computing, edge computing and big data technology.

Min Cai is a lecturer in the Faculty of information technology, Beijing University of Technology, Beijing, and a postdoctoral in computer architecture. He received his Ph.D. degree in computer application technology from Beijing Institute of Technology, Beijing, in 2014. In recent years, he has hosted and participated in a dozen of research projects in various fields such as computer architecture, artificial intelligence, smart city, industrial Internet, Internet of Things, and embedded system design, and published more than 10 papers in well-known academic journals.

Zhi Cai is an associate professor in the Faculty of Information Technology, Beijing University of Technology, Beijing. He received his M.S. degree from the School of Computer Science in the University of Manchester, Manchester, in 2007, and his Ph.D. degree in computer science and technology from the Manchester Metropolitan University, Manchester, in 2011. His research interests include information retrieval, ranking in relational databases, keyword search, and intelligent transportation systems.