Concurrency-Aware Cache Miss Cost Prediction with Perceptron Learning

Yuping Wu
wuyuping20z@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Science
University of Chinese Academy of Sciences
China

Xiaoyang Lu
xlu40@iit.edu
Department of Computer Science, Illinois Institute of Technology
USA

Xiaoming Chen
chenxiaoming@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
China

Yinhe Han
yinhes@ict.ac.cn
Institute of Computing Technology
Chinese Academy of Sciences
China

Xian-He Sun
sun@iit.edu
Department of Computer Science, Illinois Institute of Technology
USA

ABSTRACT

The widening performance gap between processors and memory has become a major bottleneck in modern computing systems, highlighting the importance of cache performance. Traditional cache replacement policies primarily exploit data locality but often neglect the critical impact of concurrency. In fact, modern caching techniques support concurrent data accesses by servicing multiple accesses concurrently. Due to concurrency, the cost of cache misses varies significantly, offering an opportunity to enhance cache replacement by prioritizing the reduction of costly misses.

In this paper, we introduce a perceptron-based concurrency-aware miss cost predictor (CAMP) to enhance locality-based cache replacement decisions. CAMP predicts the actual cost of cache misses in environments with concurrent data accesses by analyzing multiple correlated program features. Perceptron learning is used due to its lightweight and adaptable nature, enabling accurate and generalizable predictions of cache miss costs across diverse workloads. By integrating CAMP with a locality-based cache replacement policy, we demonstrate that CAMP enhances cache management for data-intensive applications by introducing concurrency awareness. Evaluations show that integrating CAMP with SHiP++ outperforms LRU by 7.1% and 12.6% in 1-core and 4-core systems on SPEC workloads, and by 10.7% in 4-core GAP workloads, surpassing state-of-the-art policies with only modest overhead.

KEYWORDS

Cache Replacement, Data Concurrency, Perceptron Learning

GLSVLSI 2025, June 30–July 2, 2025, New Orleans, LA, USA

ACM ISBN 979-8-4007-1496-2/2025/06
https://doi.org/10.1145/3716368.3735219

ACM Reference Format:
Yuping Wu, Xiaoyang Lu, Xiaoming Chen, Yinhe Han, and Xian-He Sun. 2018. Concurrency-Aware Cache Miss Cost Prediction with Perceptron Learning. In *Proceedings of Great Lakes Symposium on VLSI (GLSVLSI 2025).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3716368.3735219

1 INTRODUCTION

Over the past three decades, the growing performance gap between processors and memories has created a bottleneck where fast processors are frequently stalled, waiting for slower memory [24]. This disparity makes it crucial to enhance the performance of memory systems to improve overall computing efficiency [20]. Caches play a pivotal role in modern memory systems by mitigating memory access latency through the utilization of both data locality and concurrency [15, 17]. Beyond traditional cache hierarchies that utilize data locality, concurrency has become a prevailing aspect of cache design. Advanced caching techniques, such as multi-port [26] and non-blocking caches [14], improve data access concurrency by allowing multiple cache accesses to coexist within the same cycle.

The performance of caches is highly impacted by the underlying cache replacement policy. Traditional cache replacement policies [8, 9, 23, 25] leverage data locality to reduce cache misses. However, these locality-based replacement policies overlook data concurrency, thereby failing to utilize concurrency in cache management. In caches with multiple concurrent accesses, the cost of each cache miss varies [11, 15, 17]. The cost of a miss access can be hidden by overlapping with hit accesses or amortized by overlapping with parallel misses. Therefore, it is essential to account for concurrency to fully assess the impact of cache misses on system performance. Pure Miss Contribution (PMC) [15] is a metric used to measure the cost of each outstanding miss access by considering data access overlaps. There is potential to enhance locality-based cache replacement policies by integrating concurrency considerations, thereby tailoring the reduction of costly misses.

Previous studies [11, 15, 17] have recognized the importance of concurrency and sought to mitigate costly cache misses by retaining critical blocks. These policies identify critical blocks by predicting cache miss costs. However, these predictions rely on heuristics, and their accuracy can vary significantly depending on the workloads

and data access patterns. There is a need for intelligent techniques that can accurately predict the actual cache miss cost across various workloads. Perceptron learning [18] is a lightweight mechanism that synthesizes information from multiple features to make accurate predictions. Its adaptability allows it to dynamically adjust to diverse workloads and access patterns [3, 4, 10, 12, 22], making it ideal for predicting cache miss costs in environments with high concurrency.

In this work, we propose CAMP (Concurrency-Aware Miss Cost Predictor), a lightweight, concurrency-aware miss cost predictor based on perceptron learning, designed to enhance locality-based cache replacement policies. CAMP is not a replacement policy itself but a general predictor that provides accurate miss cost predictions across diverse workloads, enabling locality-based replacement decisions to be tuned with concurrency awareness, thereby prioritizing the retention of critical blocks. To summarize, we make the following major contributions:

(1) We propose CAMP, a perceptron-based predictor for accurate miss cost prediction in the environment with prevalent concurrency. CAMP leverages multiple correlated program features to ensure prediction accuracy.
(2) CAMP conducts online learning, adjusting its predictions based on real-time feedback. This design enables CAMP to adapt to diverse workloads and access patterns dynamically.
(3) CAMP is general and lightweight, making it practical to integrate with any traditional locality-based policies, thereby removing obstacles to its practical application.
(4) We develop a practical implementation of CAMP by using SHiP++ [25] as the underlying locality-based cache replacement policy. Our evaluations show that CAMP provides an average of 90.2% prediction accuracy and, when integrated with SHiP++, outperforms state-of-the-art policies on memory-intensive workloads.

2 BACKGROUND AND MOTIVATION

2.1 Concurrent Memory Access Model

Modern memory systems exhibit high data concurrency due to advanced cache techniques [6, 14, 26]. The Concurrent Average Memory Access Time (C-AMAT) model was proposed [21] to evaluate the impact of concurrent data accesses on performance. In C-AMAT, the cost of a cache request consists of two components: 1) base access cycles, representing the lookup time at the current cache layer, and 2) miss access cycles, the additional cycles required for misses. The C-AMAT model shows that miss access cycles can be hidden or amortized by other concurrent accesses. As illustrated in Figure 1, all miss access cycles of *Access A* are fully overlapped with base access cycles from other accesses. If there is a miss access cycle that does not overlap with any base access cycle, this cycle is called a pure miss cycle. According to C-AMAT, the memory active cycles on a memory layer are the cycles with memory activities. Active pure miss cycles refer to the memory active cycles that only contain pure miss cycles (e.g., cycles 5 to 7 in Figure 1), causing memory stalls and performance degradation.

Pure Miss Contribution (PMC) [15] quantifies the cost of cache misses by measuring their contribution to active pure miss cycles. As illustrated in Figure 1, *Access C* and *Access D* contribute to three active pure miss cycles (Cycles 5 to 7). The PMC value for *Access C* is 1, computed as $\frac{1}{2}$ from Cycle 5 plus $\frac{1}{2}$ from Cycle 6. The PMC value for *Access D* is 2, computed as $\frac{1}{2}$ from Cycle 5, $\frac{1}{2}$ from Cycle 6, and 1 from Cycle 7.

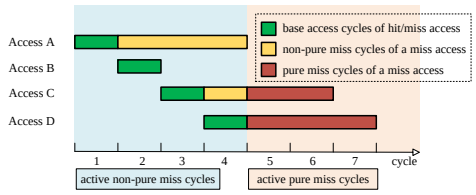

Figure 1: Illustration of C-AMAT model and PMC.

By considering data access overlaps, PMC shows that not all cache misses carry equal cost. If the PMC of each cache block can be predicted and multiple cache victim candidates are provided based on the underlying locality-based replacement policy, it becomes natural to replace the candidate block with the lowest predicted PMC. Therefore, accurate PMC prediction has the potential to complement and enhance locality-based policies.

2.2 Locality-Based Replacement Policies

Belady's OPT algorithm is the theoretically optimal locality-based replacement policy [2], evicting the block with the longest reuse distance to minimize misses. To make OPT implementable, various locality-based replacement policies are designed to predict data reuse. LRU assumes that recently used blocks are more likely to be reused. RRIP [9] predicts reuse intervals using Re-Reference Prediction Values (RRPV), which indicates the predicted time interval before the block will be referenced again. SHiP [23] (Signature-Based Hit Predictor) uses a Signature History Counter Table(SHCT) to predict and optimize the re-reference interval of caches, thereby reducing cache misses. SHiP++ [25] enhances SHiP [23] by improving cache insertion policies, refining SHCT training, and distinguishing prefetch and demand accesses. Hawkeye [8] predicts reuse distances based on access history to classify blocks as cache-friendly or cache-averse. Despite the growing complexity of data reuse prediction in locality-based policies, they assume all misses have equal cost and ignore the performance disparity caused by concurrent cache miss access. However, this assumption does not always hold true in modern memory systems with prevailing concurrent data accesses, leading to suboptimal replacement decisions.

2.3 Concurrency-Aware Replacement Policies

With the widespread presence of concurrency in memory systems, the community realizes the necessity of considering both locality and concurrency for cache optimizations [11, 15–17]. Cost-sensitive Replacement Algorithms [11] propose several LRU extension policies that consider various miss costs. However, the performance depends on accurate prediction of miss cost, which can change dynamically and affect the effectiveness of the algorithm. SBAR [17] utilizes the concurrency of miss accesses to classify cache misses into isolated misses and parallel misses. SBAR assumes that isolated misses are more costly to performance than parallel misses. SBAR uses the Memory Level Parallelism (MLP cost) to guide replacement decisions, but MLP costs can vary significantly across different program phases, affecting prediction accuracy. CARE [15] is a standalone replacement policy that considers both data reuse and PMC. However, its PMC predictions rely on heuristic, counter-based observations, assuming a strong correlation between PMC

and a single feature: the PC. The effectiveness of CARE is limited for workloads and data patterns exhibiting significant PMC variations, which restricts its overall performance. Therefore, there is an urgent need for an accurate, concurrency-aware cache miss cost predictor that can adapt to diverse data access patterns and workloads, complementing locality-based policies to provide optimized cache replacement decisions.

2.4 Perceptron Learning in Architecture Design

Perceptron learning [18] is a lightweight mechanism that synthesizes decisions by combining multiple input features. It operates by assigning weights to these features and predicting outcomes based on whether the weighted sum exceeds a threshold. This simple yet effective approach enables perceptrons to dynamically adapt to changing patterns, making them well-suited for decision-making tasks in computer architecture. Perceptron learning has been widely applied in memory optimization, particularly in cache reuse prediction [13, 22], prefetch filtering [4], and off-chip memory access prediction [3, 10]. These studies demonstrate that perceptron learning is adaptable, lightweight, and hardware-efficient, making it ideal for cache optimizations. Building on these strengths, we design CAMP, a perceptron-based PMC predictor that leverages multiple correlated program features to enhance PMC prediction accuracy. By integrating PMC-aware insights, CAMP refines cache replacement decisions, effectively balancing locality and concurrency effects.

3 CAMP DESIGN AND ARCHITECTURE

In this section, we introduce CAMP, a concurrency-aware perceptron-based cache miss predictor that provides accurate PMC range predictions for each cache block. CAMP dynamically learns correlations between multiple program features, enabling it to estimate cache miss costs more effectively than traditional heuristics. By incorporating concurrency-aware insights into locality-based replacement policies, CAMP improves cache management for data-intensive applications. While CAMP is general and applicable to any locality-based policy, we implement and evaluate it using SHiP++ [25] as our underlying policy.

3.1 PMC Measurement and Implementation

The Miss Status Holding Register (MSHR) [14] is used to track both the number of in-flight misses and the number of miss access cycles for each outstanding miss. We integrate the measurement of PMC into the MSHR entry. Suppose a miss access has n pure miss cycles. The PMC value for a specific miss access is calculated as follows:

$$PMC = \sum_{i=1}^{n} \frac{1}{N_i}, \tag{1}$$

where N_i is the number of outstanding misses in the i-th pure miss cycle for the miss access. The value of N_i is determined by monitoring the number of valid MSHR entries at each cycle. This integration provides a straightforward way to measure PMC in real-time.

Figure 2 illustrates the PMC measurement structure. The *Pure Miss Detector (PMD)* checks whether the current memory cycle is an active pure miss cycle. A 1-bit *PMD flag* is updated every cycle to indicate this status, and once an active pure miss cycle is detected, the *PMD flag* is set to 1. The *MSHR Occupation Counter*

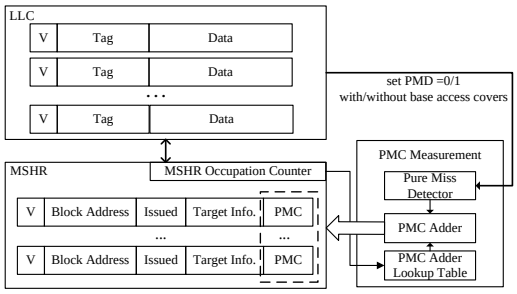

Figure 2: PMC Measurement Structure.

Table 1: Size of selected feature.

Feature	Size
Address [19 : 26]	256 × 3 bits
Address trace [7 : 15]	512 × 3 bits
PC [17 : 25]	512 × 3 bits
PC trace [25 : 33]	512 × 3 bits
PC ⊗ address [42 : 50]	512 × 3 bits
Total	0.84 KB

then counts the number of outstanding cache misses, denoted as N. To compute the PMC value for each outstanding miss, a *PMC counter* is assigned to each MSHR entry. During each active pure miss cycle, the PMC counters in all valid MSHR entries are incremented by $\frac{1}{N}$ concurrently. The number of MSHR entries determines the maximum value of N. Since the number of MSHR entries is limited (64 in our evaluation), N is an integer ranging from 1 to 64. To avoid complex floating-point computations, we precompute all possible values of $\frac{1024}{N}$ and store them in a rounded integer format in a *PMC Adder Lookup Table*. The *PMC Adder* accumulates the sum as $1024 \times$PMC, and the final result is bit-shifted right by 10 to compute the actual PMC.

Figure 3 shows the distribution of PMC values across various SPEC workloads, revealing significant variation among benchmarks. For example, the PMC values of `473` and `649` are concentrated in higher intervals, whereas those of `433` and `623` tend to be lower. Meanwhile, `603` and `605` exhibit a more uniform distribution. This variation in PMC distribution highlights that cache misses have diverse impacts on performance, suggesting opportunities to prioritize mitigating misses with high PMC values.

3.2 CAMP Design and Workflow

We design CAMP using a perceptron learning model to predict whether the PMC value of a cache miss exceeds the average PMC value. CAMP is organized as a collection of weight tables, each corresponding to a selected feature. These weight tables store values using 3-bit saturating counters, representing the correlation between the features and the predicted PMC interval. The size of selected features are listed in Table 1. The overall workflow of CAMP, including its prediction mechanism, is illustrated in Figure 4.

On an LLC miss, CAMP first trains its model and then predicts the PMC interval of the incoming block. During training, CAMP extracts a set of program features from the current load request and selects the most relevant bits of each feature to index the corresponding weight tables. It then retrieves and sums the associated weights to compute a cumulative perceptron weigh (W_{train}). This value is used to predict whether the PMC value of the block is greater than the average PMC value. The predefined threshold, denoted as θ, guides the prediction decision. If $W_{train} < \theta$, the PMC

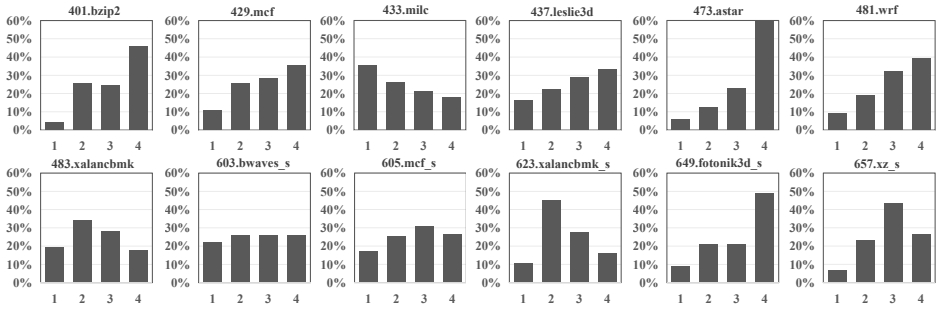

Figure 3: Coarse-grained PMC distribution. (The x-axis represents the PMC value intervals in cycles: 1 = 0-39 cycles; 2 = 40-79 cycles; 3 = 80-179 cycles; 4 = 180+ cycles.)

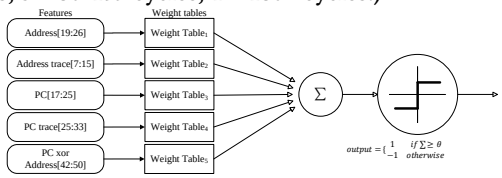

Figure 4: Workflow of CAMP.

Table 2: Program features for PMC prediction.

Feature	Description
Address	The virtual address of current access
Address trace	The XOR value of last-4 addresses
PC	Program Counter of current access
PC trace	The XOR value of last-4 PC
PC ⊗ Address	The XOR value of PC and address
page number	The page number of current access
page offset	The page offset of current access
cache set number	The cache set number of current access

of the incoming block is predicted to be lower than the average PMC value. Otherwise, the PMC value is classified as high. We tested various thresholds and ultimately set θ to 18. As discussed in Section 3.1, the actual PMC value of the block is measured in parallel. CAMP compares this measured PMC with the average PMC to verify the prediction's correctness. If the prediction is incorrect, CAMP updates the weight values for each feature: if the actual PMC is greater than the average PMC, the weights are incremented, otherwise, the weights are decremented. Once training is completed, CAMP recomputes the cumulative perceptron weight (W) based on the updated tables and predicts the future PMC interval, assessing the criticality of the incoming block. Each cache block is configured with a 1-bit *PMCL* flag, which records whether the predicted PMC is above the average value. *PMCL* is set to 1 if the predicted PMC of the cache access exceeds the average PMC. In practical implementation, once the weight tables are updated, the new weights are retrieved directly without re-indexing, eliminating redundant lookup overhead. We dynamically update the average PMC value over a fixed number of memory accesses. The measurement interval is set to half of the LLC capacity, meaning the average PMC update occurs after half of the LLC blocks are evicted.

3.3 Selection of Features

Unlike simple counter-based schemes, perceptron learning dynamically adapts across workloads by leveraging multiple features and capturing their correlations, resulting in improved prediction accuracy. The selection of correlated features is critical for designing an accurate PMC predictor. The feature selection process is performed offline during the design phase of CAMP. We evaluate a range of

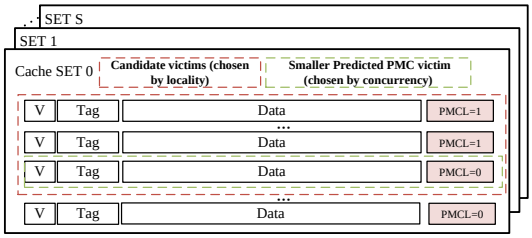

Figure 5: Integrating CAMP with SHiP++.

candidate program features that may influence PMC and analyze their correlation by measuring their individual prediction accuracy, as detailed in Table 2. To refine the selection, each program feature is partitioned into multiple subcomponents, represented as bit ranges from S to E ($[S : E]$). The prediction accuracy of each subcomponent is evaluated independently to determine its contribution to PMC prediction. The highest-performing subcomponent is selected as the representative of its corresponding feature. Finally, only representatives with a prediction accuracy exceeding 70% are retained as final predictive features, as summarized in Table 1.

`Address` values tend to cluster within contiguous address spaces, such as arrays and vectors, particularly within a given program phase. Accesses with similar address bits often exhibit similar data concurrency patterns. `PC` identifies the instruction responsible for initiating a data load or store. In many programs, memory accesses initiated by the same PC exhibit similar concurrency characteristics, making PC a strong feature for PMC prediction. `Address trace` and `PC trace` capture longer-term access patterns across different program phases, allowing the predictor to make more accurate predictions. The combined feature `PC` ⊗ `Address` considers both PC and address together, providing fine-grained contextual information that captures the interaction between program behavior and PMC characteristics.

3.4 Integrating CAMP with SHiP++

Figure 5 illustrates how CAMP cooperates with locality-based cache replacement policies to make replacement decisions. When the cache needs to select a block for replacement, we first identify candidate victims based on locality. Then, CAMP is used to choose the block with the PMCL-bit set to 0 among candidates exhibiting poor locality for eviction. In this implementation, both locality and concurrency information are considered when making eviction decisions, providing a more comprehensive view of cache management. Note that the design of CAMP is general and can be

Table 3: Simulator configurations.

Processor	1-core and 4-core, 4GHz, 8-issue width, 352-entry ROB, 128 Load Queue, 72 Store Queue, Perceptron branch predicton [12]
L1 Cache	private, 32 KB I/D-cache per core, 8-way, 4-cycle latency, 8/16-entry MSHR, LRU, next-line prefetcher
L2 Cache	private, 256 KB per core, 8-way, 8-cycle latency, 32-entry MSHR, LRU, IP-stride prefetcher
L3 Cache	shared, 2MB per core, 16-way, 20-clcle latency, 64-entry MSHR
DRAM	4 GB, tRP = 15ns, tRCD=15ns, tCAS = 12.5ns

Table 4: Evaluated workloads.

SPEC06	401.bzip2, 410.bwaves, 429.mcf, 433.milc, 434.zeusmp, 437.leslie3d, 450.soplex, 459.GemsFDTD, 462.libquantum, 473.astar, 481.wrf, 482.sphinx3, 483.xalancbmk
SPEC17	603.bwaves_s, 605.mcf_s, 607.cactuBSSN_s, 623.xalancbmk_s, 649.fotonik3d_s, 654.roms_s, 657.xz_s
GAP	bc, bfs, sssp, tc

integrated into most locality-based cache replacement policies using the same workflow. For other locality-based cache replacement policies, CAMP can also leverage the PMCL-bit in a similar way.

We present a case study implementation of CAMP, which provides the predicted PMCL for each incoming block to introduce concurrency awareness into the cache replacement decisions of locality-based policies. In this case, we select SHiP++ [25] as the underlying locality-based policy. SHiP++ is a well-designed locality-based cache policy that predicts RRPVs for cache replacement. When an incoming block arrives, SHiP++ iterates through the corresponding cache set and selects the first cache block encountered with the maximum preset RRPV value as the victim. After integrating with CAMP, cache eviction further incorporates the PMCL-bit, ensuring that among blocks with poor locality, the one with the lowest predicted PMC is prioritized for eviction.

4 EXPERIMENT AND RESULTS

4.1 Methodology

We evaluate CAMP using the ChampSim [7] simulator. Table 3 details our evaluation system configuration. To better align with practical cache architectures, we incorporate a next-line prefetcher at L1 and an IP-stride prefetcher at L2. We select LRU as the baseline for performance comparison and evaluate CAMP as an enhancement to SHiP++ [25][1], analyzing its impact on cache management compared to three state-of-the-art LLC cache replacement policies: SHiP++ [25], Hawkeye [8], and CARE [15]. We evaluate CAMP using 20 memory-intensive workloads from SPEC 2006 [19] and SPEC 2017 [5], along with 4 from GAP [1], all of which have at least 1 LLC miss per kilo instruction (MPKI) under the LRU policy, as shown in Table 4. For SPEC workloads in a 4-core system, we evaluate performance on both homogeneous and heterogeneous workload mixes. For homogeneous mixes, we run 4 identical copies of the same workload. For heterogeneous mixes, we randomly select 4 different workloads per core and generate 25 mixed workloads. For GAP workloads, we evaluate performance using homogeneous mixes of the same workloads in a 4-core system.

4.2 Prediction Accuracy of CAMP

We evaluate the efficiency of CAMP by measuring the accuracy of PMC predictions across homogeneous mixes of 20 memory-intensive SPEC workloads in a 4-core system. Figure 6 shows that

[1]In Section 4.3, 'CAMP' refers to the integration of CAMP with SHiP++, where CAMP enhances SHiP++ by incorporating PMC-aware predictions.

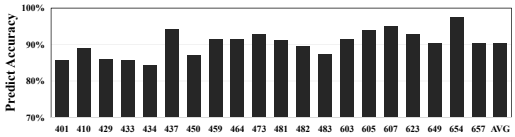

Figure 6: Prediction accuracy of CAMP.

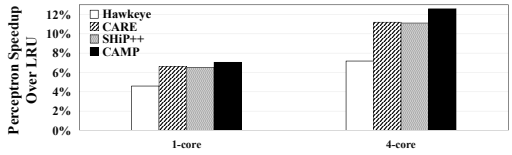

Figure 7: Average Speedup for 1-core and 4-core (homogeneous mixes) on SPEC workloads.

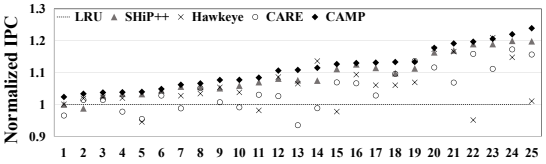

Figure 8: Normalized IPC for 4-core (heterogeneous mixes) on SPEC workloads.

the prediction accuracy exceeds 84.4% for all workloads, with an average of 90.4% and a peak accuracy of 97.5%. These results demonstrate that CAMP reliably predicts whether the PMC value exceeds the average, effectively capturing data access concurrency and providing a strong foundation for LLC management.

4.3 Performance Comparison

Figure 7 shows the speedup for 1-core and 4-core systems on SPEC workloads as listed in Table 4. In the 1-core system, CAMP (integrate with SHiP++) outperforms LRU by 7.1%, while Hawkeye, CARE and SHiP++ achieve average performance improvements of 4.6%, 6.6% and 6.5%, respectively. By incorporating both data locality and concurrency, CAMP surpasses locality-based policies like SHiP++ and Hawkeye. Compared to CARE, which also considers data concurrency in LLC management but relies on heuristic, counter-based PMC prediction, CAMP offers a significant advantage with its adaptive PMC prediction. In the 4-core system with SPEC homogeneous workload mixes, CAMP outperforms LRU by 12.6%, while Hawkeye, CARE and SHiP++ improve performance by 7.2%, 11.2%, and 11.1%, respectively on average. We observe that CAMP exhibits significant advantages as the number of CPU cores increases, benefiting from higher data concurrency in multi-core systems.

Figure 8 shows the normalized weighted speedup over LRU for 4-core heterogeneous mixed workloads on SPEC workloads. Weighted speedup is a widely used metric for evaluating shared cache performance [8, 15]. CAMP improves performance over LRU by 11.0%, while SHiP++, Hawkeye, and CARE improve performance by 9.1%, 5.4%, and 4.4%, respectively. CAMP outperforms all competing policies on 22 of the 25 mixed workloads and is slightly sub-optimal on the remaining three. These results demonstrate CAMP's robustness in handling complex multi-core workloads.

Figure 9 shows the normalized IPC among homogeneous GAP workloads in a 4-core system. CAMP outperforms LRU by 10.7%, while Hawkeye, CARE and SHiP++ achieve performance improvements of 3.7%, 10.1%, and 7.4%, respectively. We observe that CAMP

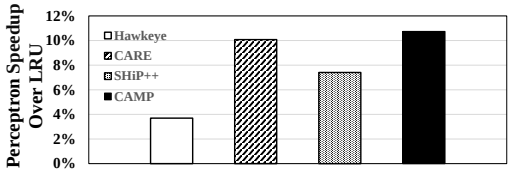

Figure 9: Average Speedup for 4-core (homogeneous mixes) on GAP workloads.

Table 5: Hardware cost of CAMP.

Portion	Size	Used for
PMCL	4KB	Record whether predicted PMC is large
Weight tables	0.84KB	Perceptron training and prediction
PMC adder lookup table	0.25KB	PMC measurement
Total: 5.09KB for concurrency awareness		

Table 6: Hardware costs for different replacement policies.

Replacement Policy	Concurrency-Aware	Total Cost
LRU	No	16KB
SHiP++ [25]	No	16KB
Hawkeye [8]	No	30.94KB
CARE [15]	Yes	26.64KB
CAMP+SHiP++	Yes	**21.09KB**

and CARE significantly surpass the locality-based SHiP++ and Hawkeye on graph workloads due to their consideration of data concurrency. Integrating CAMP significantly enhances SHiP++, improving the performance of SHiP++ by 3.3% and making it more adaptive to both locality and concurrency effects. This enhancement allows it to further surpass the concurrency-aware CARE, demonstrating the effectiveness of CAMP.

4.4 Hardware Overhead of CAMP

Table 5 shows the hardware cost of CAMP for a 16-way 2MB LLC. For each cache block allocated in the LLC, we assign a 1-bit PMCL, resulting in a total of 4KB. For the perceptron learning, we utilize a 3-bit counter for each feature weight table, with a total cost of 0.84KB. To measure the actual PMC of each incoming cache block, a PMC adder lookup table is used to avoid complex computations, costing 0.25KB in total. Therefore, CAMP requires only 5.09KB for PMC measurement, perceptron training, and PMC predictions, which is lightweight at just 0.25% of a 2MB LLC. Since CAMP is a predictor rather than a full replacement policy, it is designed to be lightweight, making it practical for real-world integration with existing locality-based policies.

To put this into perspective, we compare the hardware costs of all state-of-the-art policies, as shown in Table 6. The integration with SHiP++ adds an additional 16KB to the hardware cost, bringing the total implementation cost to 21.09KB. This incurs significantly lower overhead than Hawkeye and CARE while delivering more stable and efficient cache management across diverse workloads. This demonstrates that CAMP's lightweight design and efficient PMC-aware predictions make it a highly practical and cost-effective enhancement for modern locality-based cache replacement policies.

5 CONCLUSION

In this paper, we introduce CAMP, a concurrency-aware cache miss cost predictor. CAMP accounts for all types of data access overlaps and employs lightweight perceptron learning to provide accurate miss cost predictions for each incoming cache block. We present a case study implementation of CAMP using SHiP++ as the underlying locality-based replacement policy. Extensive evaluations show that by fully incorporating both data locality and concurrency in cache management, CAMP effectively improves replacement decisions and demonstrates strong potential for data-intensive workloads.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China under Grant 62488101, Grant 62495104, and Grant 62025404, and in part by Youth Innovation Promotion Association CAS.

REFERENCES

[1] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[2] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.
[3] Rahul Bera et al. 2022. Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction. In *55th MICRO*. 1–18.
[4] Eshan Bhatia, Gino Chacon, et al. 2019. Perceptron-based prefetch filtering. In *the 46th ISCA*. 1–13.
[5] James Bucek et al. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ICPE*. 41–42.
[6] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *the 11th ICS*. 68–75.
[7] Nathan Gober et al. 2022. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324* (2022).
[8] Akanksha Jain and Calvin Lin. 2016. Back to the future: leveraging Belady's algorithm for improved cache replacement. In *43rd ISCA*. IEEE, 78–89.
[9] Aamer Jaleel et al. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH CAN* 38, 3 (2010), 60–71.
[10] Alexandre Valentin Jamet et al. 2024. A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering. In *2024 HPCA*. 528–542.
[11] J. Jeong and M. Dubois. 2003. Cost-sensitive cache replacement algorithms. In *The Ninth HPCA, 2003.* 327–337.
[12] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *the 7th HPCA*. IEEE, 197–206.
[13] Daniel A Jiménez and Elvira Teran. 2017. Multiperspective reuse prediction. In *the 50th MICRO*. 436–448.
[14] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *the 8th ISCA*. IEEE Computer Society Press, 81–87.
[15] Xiaoyang Lu et al. 2023. CARE: A concurrency-aware enhanced lightweight cache management framework. In *2023 HPCA*. IEEE, 1208–1220.
[16] Xiaoyang Lu et al. 2024. CHROME: Concurrency-aware holistic cache management framework with online reinforcement learning. In *2024 HPCA*. IEEE, 1154–1167.
[17] Moinuddin K Qureshi and ohters. 2006. A case for MLP-aware cache replacement. *ACM SIGARCH CAN* 34, 2 (2006), 167–178.
[18] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.
[19] Cloyce D. Spradling. 2007. SPEC CPU2006 benchmark tools. *SIGARCH Comput. Archit. News* 35, 1 (mar 2007), 130–134.
[20] Xian-He Sun and Xiaoyang Lu. 2023. The memory-bounded speedup model and its impacts in computing. *JCST* 38, 1 (2023), 64–79.
[21] Xian-He Sun and Dawei Wang. 2013. Concurrent average memory access time. *Computer* 47, 5 (2013), 74–80.
[22] Elvira Teran, Zhe Wang, and Daniel A Jiménez. 2016. Perceptron learning for reuse prediction. In *2016 MICRO*. IEEE, 1–12.
[23] Carole-Jean Wu et al. 2011. SHiP: Signature-based hit predictor for high performance caching. In *the 44th MICRO*. 430–441.
[24] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH CAN* 23, 1 (1995), 20–24.
[25] Vinson Young et al. 2017. SHiP++: Enhancing signature-based hit predictor for improved cache performance. In *CRC held in conjuction with ISCA*.
[26] Zhaomin Zhu et al. 2003. A novel hierarchical multi-port cache. In *the 29th ESSCIRC*. IEEE, 405–408.