CARE: A Concurrency-Aware Enhanced Lightweight Cache Management Framework

Xiaoyang Lu
Department of Compute Science
Illinois Institute of Technology
Chicago, Illinois
xlu40@hawk.iit.edu

Rujia Wang
Department of Compute Science
Illinois Institute of Technology
Chicago, Illinois
rwang67@iit.edu

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

***Abstract*—Improving cache performance is a lasting research topic. While utilizing data locality to enhance cache performance becomes more and more difficult, data access concurrency provides a new opportunity for cache performance optimization. In this work, we propose a novel concurrency-aware cache management framework that outperforms state-of-the-art locality-only cache management schemes. First, we investigate the merit of data access concurrency and pinpoint that reducing the miss rate may not necessarily lead to better overall performance. Next, we introduce the pure miss contribution (PMC) metric, a lightweight and versatile concurrency-aware indicator, to accurately measure the cost of each outstanding miss access by considering data concurrency. Then, we present CARE, a dynamic adjustable, concurrency-aware, low-overhead cache management framework with the help of the PMC metric. We evaluate CARE with extensive experiments across different application domains and show significant performance gains with the consideration of data concurrency. In a 4-core system, CARE improves IPC by 10.3% over LRU replacement. In 8 and 16-core systems where more concurrent data accesses exist, CARE outperforms LRU by 13.0% and 17.1%, respectively.**

I. Introduction

Intensive research has been conducted to address the memory wall problem [50], of which improving locality and concurrency are two fundamental approaches. Cache hierarchies utilize data locality to minimize the long delay of off-chip main memory accesses. Significant research focuses on taking advantage of data locality, resulting in many schemes that detect memory access patterns, so that cache eviction and insertion decisions can be determined by the reference predictions to reduce cache miss rate [8], [11], [13], [14], [15], [17], [18], [19], [20], [21], [23], [25], [26], [27], [33], [35], [36], [38], [39], [41], [45], [47], [48], [51], [53]. Although such locality-based cache management frameworks may reduce the number of misses, we find that considering both locality and concurrency can further improve the state-of-the-art locality-only optimizations.

Modern high-performance processors support data access concurrency [12] with advanced caching techniques such as multi-port, multi-bank, pipelined, and non-blocking cache. As a result, multiple outstanding cache accesses can be generated by one processor and overlapped with each other. With data access concurrency in the memory hierarchy, the cost of a miss could vary. Some misses are isolated, some misses occur concurrently with other hits, and some misses overlap with other miss accesses [29]. The performance loss resulting from a cache miss can be hidden by access overlapping. Thus, a more accurate cost metric for cache misses may help improve cache performance further when data concurrency and overlapping exist [30], [34], [44], [52].

In this work, we first introduce and formally define the concept of *Pure Miss Contribution (PMC)*. PMC is a new cost metric for cache misses, with a comprehensive analysis of both hit-miss and miss-miss overlapping in the memory system. PMC has high predictability and versatility. We observe that the PMC values of the misses caused by the same program counter (PC) are relatively stable; therefore, the past PMC value can be used to predict the future PMC value of the same load instruction. PMC is also lightweight to measure and versatile enough to be used to build concurrency-aware cache management frameworks. We then present CARE, a concurrency-aware cache management framework that takes both data locality and concurrency into account. CARE learns the re-reference behavior and PMC value of each miss access to guide future replacement decisions. CARE augments existing cache insertion and hit-promotion policies to reserve a small subset of performance-critical blocks with high locality and high PMC, and evict dead blocks or blocks with low PMC. CARE is also prefetch-aware, and it performs well under prefetchers. In CARE, we also implement a Dynamic Threshold Reconfiguration Mechanism (DTRM), which enables CARE to better adapt to different applications and execution phases. Our experimental results show that CARE outperforms state-of-the-art cache management schemes. Furthermore, CARE has low overhead and can be practically implemented in hardware. To summarize, this paper makes the following contributions:

1) We introduce the pure miss contribution (PMC), a novel and accurate metric to quantify the cost and performance impact of outstanding cache misses. We describe how PMC can be measured in modern cache hierarchies. We find that PMC is predictable and can be used for cache optimization.
2) We present CARE, a comprehensive cache management framework that considers both locality and concurrency. CARE is general for all types of applications, practical with low hardware implementation overhead, and adaptive with a novel Dynamic Threshold Reconfiguration Mechanism (DTRM).

3) Our evaluations show that CARE substantially improves upon existing state-of-the-art cache management schemes over a wide variety of workloads in a wide range of system configurations and performs well with data prefetching.

II. Background and Preliminaries

A. Memory Level Parallelism

Multi-core and multi-threading designs, as well as advanced caching techniques [7], [13], [24], [34], [37], increase data access concurrency. As a result, a number of memory accesses can concurrently coexist in the memory hierarchy. In this case, some memory accesses may overlap with others, which reduces their performance impact on cores.

Memory Level Parallelism (MLP) can be used to measure miss concurrency. MLP captures the number of outstanding cache misses that can be generated and executed in an overlapped manner [16]. Some misses are isolated, while some occur concurrently with other misses. The more cache misses occur concurrently, the smaller the impact of each cache miss on performance since all concurrent misses will amortize the total memory stall cycles. Therefore, based on the MLP concept, isolated misses are considered to hurt performance more than concurrent misses. MLP can be measured with MLP-based cost [34]. The MLP-based cost of an isolated miss can be approximated by the number of miss-access cycles that the miss spends. For concurrent misses, the data access delay is divided equally among all concurrent outstanding misses, representing the MLP-based cost of each concurrent miss access.

MLP-based cost can identify costly misses by considering the miss-miss overlapping. However, we find that hit-miss overlapping impacts the cost of misses as well, and modern memory systems have a lot of such overlapping accesses (details in Section III-B). Therefore, we need a holistic metric that is able to catch all types of overlapping and provide a better memory performance optimization guidance.

B. Concurrent Memory Access Model

To capture all types of concurrent memory accesses and quantify their impact on performance, a concurrent memory access model named C-AMAT was proposed [44]. In the C-AMAT performance model, a cache access latency is composed of two parts: 1) base access cycles, which are the minimum time an access (hit or miss) needs to spend on a specific cache level; 2) miss access cycles, which are the additional time spent waiting for data in the next levels of the memory hierarchy. For a miss access, the tag lookup time is considered to be the base access cycles of the access. A miss access latency consists of both base access cycles and miss access cycles. Figure 1 shows several concurrent cache accesses. Each access spends two base access cycles, and each miss access consumes three additional miss access cycles.

Based on the C-AMAT model, the miss access cycles can be hidden when there is a hit-miss overlapping.[1] Therefore the actual cost of the misses should be revisited. On the other side, for a miss access cycle that does not overlap with any base access cycle, we refer to this cycle as a *pure miss cycle* [28], [29], [30], [31]. If a miss access contains at least one pure miss cycle, this miss is categorized as *pure miss*. Pure miss has a higher performance impact because pure miss cycles have no overlapping base access cycles to hide the penalty. Similar to miss rate, the *Pure Miss Rate (pMR)* can measure the cache efficiency by considering data access concurrency. The formal definition of pMR is as below:

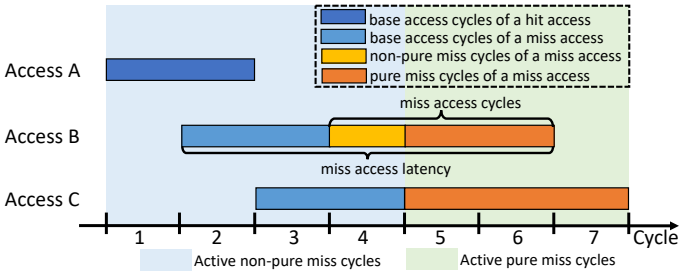

Fig. 1: Illustration of C-AMAT model and Pure Miss.

$$\text{Pure Miss Rate (pMR)} = \frac{\text{Num. of Pure Misses}}{\text{Num. of Total Accesses}}$$

Based on the C-AMAT model, the *memory active cycles* on a memory layer are the cycles with memory activities [28]. Active miss cycles are classified into two categories: *active pure miss cycles* and *active non-pure miss cycles*. Active pure miss cycles are the cycles that only contain the pure miss cycles (cycles 5 to 7 in Figure 1), and these cycles cause more performance degradation. On the other hand, the active non-pure miss cycles do not introduce heavy degradation, as the miss access cycle is overlapped and hidden (cycle 4 in Figure 1).

The C-AMAT concurrent memory access model is general and can be applied to each level of the memory hierarchy. In multi-core systems, the model works by tracking the overlapping from each core. In other words, the pure miss in a multi-core system contains at least one miss access cycle without any overlapped base access cycles from the *same core* that overlaps with. We find that the concepts in C-AMAT can capture all types of memory access overlapping. If we can quantify the cost of memory misses with all types of overlapping, we can use the metric to enable cache optimization further. In this work, we present pure miss contribution metric (details in Section IV), which is inspired by C-AMAT model, and it shows the great potential to be incorporated with cache optimization frameworks.

C. Locality-based Cache Management

Locality-based cache management schemes are designed to increase performance by reducing the total number of misses. The ideal upper bound for such schemes is Belady's optimal replacement (OPT) [10], which always evicts the block with the largest future usage distance. Recent locality-based cache

[1] It refers to miss access cycles overlapped with the base access cycles of a hit/miss access.

management studies have focused on exploring prediction-based schemes to reduce the number of cache misses [17], [19], [40], [42], [45], [48].

Re-reference prediction. Several replacement studies are designed based on the re-reference prediction of cache blocks, determining the lifetime of the blocks in the cache. SRRIP [19] statically predicts an "intermediate" re-reference interval at cache insertion time and updates the re-reference prediction on subsequent accesses. DRRIP [19] is proposed to improve performance by selecting the inserting position among different policies. Recent studies exploit long-term information to increase prediction accuracy by analyzing the cache blocks that have been evicted. SHiP [48] and SHiP++ [53] provide a finer granularity re-reference prediction by correlating the re-reference behavior of cache blocks to the PCs and learning the past behavior of SRRIP. SHiP uses a history table (SHCT) to learn the re-reference characteristic for each signature. SHCT updates on cache hits and block evictions. The re-reference characteristic of each incoming block is predicted by indexing it into the SHCT. SHiP++ enhances SHiP re-reference predictions and SHCT training, further improving the last-level cache hit rate on SHiP. Jain and Lin propose Hawkeye [17]. Hawkeye simulates and learns Belady's optimal solution for a long history of memory accesses to predict the re-reference characteristic of future accesses. Hawkeye formulates the re-reference prediction as a binary prediction problem. If the incoming block is predicted to be "cache-friendly", it will be inserted with a high priority. Otherwise, "cache-averse" blocks will be marked as eviction candidates. Following in the footsteps of Hawkeye, Mockingjay [41] mimics Belady's optimal solution effectively and introduces a cache replacement policy based on multi-class re-reference prediction.

Machine learning for re-reference prediction. In recent years, machine learning is also widely used to increase the effectiveness of cache management. Teran et al. [45] propose using perceptron learning for re-reference prediction. Perceptron learning can find independent correlations between multiple input features related to block re-reference, guaranteeing accurate re-reference prediction. Glider [42] uses an offline attention-based long short-term memory (LSTM) model to improve prediction accuracy and gain insights. Then these insights are fed into an Integer Support Vector Machine (ISVM) that matches the LSTM's prediction accuracy. While machine learning models such as perceptron and ISVM can be trained online, this requires the involvement of a large number of prediction tables, which imposes a non-negligible overhead, especially in multi-core systems. Sethumurugan et al. [40] use reinforcement learning to learn a cache replacement policy. Based on the insights derived from the neural network, a cost-effective cache replacement policy RLR is proposed.

The overhead of machine learning-based techniques is difficult to justify, including the training overhead, computation cost, and model size. Our design focuses on efficiency and lightweight, which is more practical to be implemented on latency-critical cache hierarchy without any pre-processing overhead.

D. Cost-based Cache Management

Unlike locality-based cache management schemes that focus on reducing cache misses, several works improve cache performance by selectively eliminating expensive misses. LACS [22] is proposed based on the observation that the more instructions the processor issues during the miss, the more likely it is to hide the penalty for that miss. Consequently, LACS utilizes the number of instructions issued during an LLC miss to estimate the miss cost. While simple, the cost estimation model of LACS is not cycle-accurate, it is impossible to estimate the penalty of the misses on performance accurately. The MLP-aware cache replacement policy SBAR [34] takes into account the concurrency of the cache misses and observes that some misses occur alone while some occur concurrently with others. It improves performance by reducing the number of costly isolated misses. However, as we discussed in Section II-A, MLP does not consider hit-miss overlapping. Therefore, the MLP-aware replacement policy can be further improved.

III. Motivation

A. The Limitations of Locality-based Cache Management

Modern mainstream processors contain many cores and run different applications concurrently. Therefore, the shared LLC can observe very mixed access patterns. The mixed access patterns can downgrade the effectiveness of locality-based schemes since most of them make predictions only dependent on one specific access pattern. In addition, at the LLC, the recency-friendly access patterns get filtered by upper-level caches, which makes it even harder to directly get benefits from locality-based schemes [48], [49]. On the other hand, locality-based schemes all have a simple optimization goal: reducing the number of misses. It works fine for sequential memory accesses but could be better for handling prevailing concurrent cache/memory activities.

B. The Limitations of MLP-based Cache Management

In a scalable system with memory access concurrency, not all cache misses will have the same impact on performance [30], [34]. Eliminating isolated misses (high-cost) helps performance more than eliminating concurrent misses (low-cost). According to the above assumption, an MLP-aware cache replacement algorithm [34] was proposed to reduce the number of high-cost misses.

We introduce the definition of MLP-based cost in Section II-A. We show how MLP-based cost is calculated in a single-core system using the study case in Figure 2. The case in a multi-core system is similar since we only analyze the memory concurrency coming from each core independently. Here, B, F are hits; A, C, D, and E are misses. Each access consumes two base access cycles, and each miss access has six additional miss access cycles. All the accesses are at the same cache level in the memory hierarchy.

When considering the definition of MLP-based cost, access A is a miss with the highest MLP-based cost. Because there are no miss access cycles from other misses that overlap with A's miss access cycles from cycle 3 to cycle 6, and the miss access

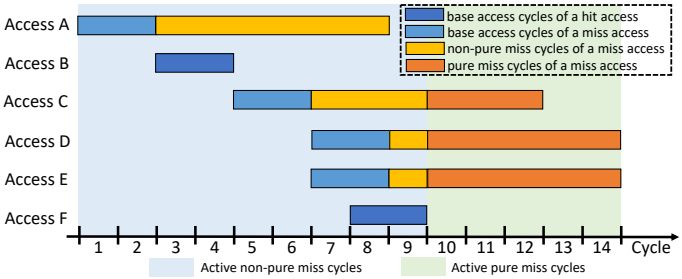

Fig. 2: Study case of concurrent memory accesses from a single application.

TABLE I: MLP-based cost analysis of the study case.

Miss	MLP-based cost
A	1 + 1 + 1 + 1 + 1/2 + 1/2 = 5
C	1/2 + 1/2 + 1/3 + 1/3 + 1/3 + 1/3 = 7/3
D	1/3 + 1/3 + 1/3 + 1/3 + 1/2 + 1/2 = 7/3
E	1/3 + 1/3 + 1/3 + 1/3 + 1/2 + 1/2 = 7/3

cycles of access A only overlap with the miss access cycles of access C in cycle 7 and cycle 8. Therefore, the MLP-cost of access A is 5. From cycle 7 to cycle 8, the miss access cycles of access C overlap with the miss access cycles of access A, so the MLP-based cost of access C in these two cycles is 1 $(1/2 \times 2)$. In addition, the miss access cycles of C still overlap with D's and E's miss access cycles from cycle 9 to cycle 12. Therefore, the MLP-based cost of access C in these four cycles is 4/3 $(1/3 \times 4)$. To sum up, the MLP-based cost of access C is 7/3. D and E have a similar situation to C, and their MLP-based costs are also 7/3. The quantitative MLP-based cost analysis is summarized in Table I.

If we re-evaluate the misses in this case study with the C-AMAT-based model, only access C, D, and E are pure misses. Access C has three pure miss cycles (cycles 10-12), and both access D and E have five pure miss cycles (cycles 10-14). Even though the MLP-based cost of access A is the highest, it does not hurt the performance the most. All of the miss access cycles of A are overlapped with base access cycles of B, C, D, E, and F. Furthermore, although the MLP-based costs of access C, D, and E are the same, they have different pure miss cycles and have various contributions to the total active pure miss cycles. This is because that MLP-based cost calculates the cost by analyzing the miss-miss overlapping but does not consider hit-miss overlapping.

Figure 3 illustrates the percentage of LLC misses that have hit-miss overlapping from the *same core* for a set of 4-core multi-copy workloads with LRU policy (Section VI details the methodology). In all benchmarks, 30% to 80% misses have hit-miss overlapping. Therefore, in order to accurately quantify the cost of cache misses, hit-miss overlapping cannot be ignored. Although LLC pure misses do not directly cause CPU stalls, LLC pure misses can seriously increase the latency of providing data to the upper-level caches. In this work, we develop a concurrency-aware enhanced cache management framework for LLC to eliminate the costly pure misses and improve the performance.

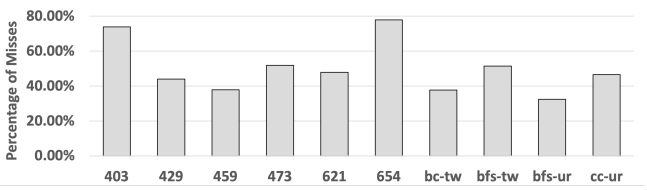

Fig. 3: Percentage of misses with hit-miss overlapping.

IV. Pure Miss Contribution

A. Definition

In this section, we introduce *Pure Miss Contribution (PMC)*, which is a new metric that considers the integrated influence of locality, concurrency, and overlapping of memory accesses. PMC recognizes that not all outstanding cache misses have the same cost, and identifies high-cost cache misses to better performance optimization.

PMC is defined as the contribution of each miss to the total active pure miss cycles from the *same core*. Consequently, PMC can be used to quantify the performance impact of each cache miss. A pure miss access has at least one pure miss cycle, which can significantly hurt the performance. Therefore, a pure miss contributes to active pure miss cycles with a positive PMC value. On the contrary, if a cache miss is not a pure miss and all its miss access cycles are overlapped with other base access cycles, its PMC value is 0.

B. Measurement and Implementation

The algorithm to detect and measure PMC in a multi-core system is described in Algorithm 1. It can be applied at each level of a memory hierarchy. In particular, we use a parameter l to indicate the specific cache level under consideration. We declare the bit and field used for PMC calculation on top of Algorithm 1. The *NoNewAccess_x* bit is used to identify the current cycle status of cache level l for core x. If there is a new cache access from core x at cache level l, *NoNewAccess_x* is reset to 0 for base access cycles. Therefore, when *NoNewAccess_x* is 1, it means that there are no base access cycles in any type of accesses that can be used to hide miss access cycles.

Modern memory systems manage data in cache blocks and utilize the Miss Status Holding Register (MSHR) [46] to handle concurrent cache misses. An MSHR entry is allocated for a miss access when the tag search fails to match anything, which is during the base access cycles of a miss. MSHR can track the information of all outstanding cache misses. A new field PMC is added for each MSHR entry to calculate PMC.

When a miss access allocates an MSHR entry, the PMC counter associated with the miss will be initialized to zero. *NoNewAccess_x* bit is used to determine whether this cycle is an active pure miss cycle for core x. If *NoNewAccess_x* is set, all outstanding misses from core x in the MSHR are pure misses that contribute to core x's active pure miss cycles. Therefore, we only calculate and update the PMC of each cache block in active pure miss cycles. For concurrent pure misses, the active pure miss cycle can be evenly divided among all

Algorithm 1 Measure PMC for cache misses at cache level l (called every active memory cycle)

NoNewAccess_x: single-bit cycle status identifier per core; set if no overlapping opportunites in this cycle;
PMC: field in MSHR to calculate the pure miss contribution for a miss block; initialized to 0;
update(N_x): check the number of outstanding misses from core x at cache level l in this cycle.

```
for x-th core in the system do
    if NoNewAccess_x is set then
        update(N_x);
        for i-th outstanding miss in MSHR do
            MSHR[i].PMC += 1/N_x;
        end for
    end if
end for
```

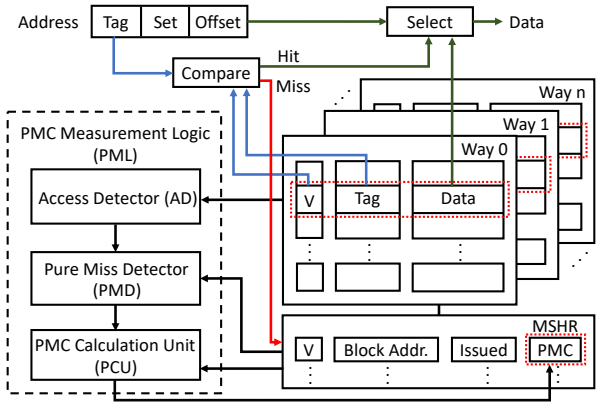

Fig. 4: PMC Measurement Structure.

concurrent pure misses. Let N_x be the number of outstanding misses from core x at cache level l in the corresponding MSHR. For every active pure miss cycle, the PMC counter of each outstanding miss in MSHR is incremented by $1/N_x$ until the requested data is serviced. Please note that PMC measurement in a single-core system is a special case of multi-core (the number of cores is 1).

Figure 4 illustrates the PMC measurement logic (PML), which follows Algorithm 1 to record PMC values of outstanding misses. The Access Detector (AD) can detect base access cycles and notify the Pure Miss Detector (PMD) whether the current cycle is an active pure miss cycle. The base access cycles are known and fixed for any given cache level. Therefore, the AD monitors for a fixed amount of cycles and sets the *NoNewAccess* bit accordingly.

Using the information collected from the AD and the miss information from the MSHR, PMD can identify the pure miss accesses. PMC Calculation Unit (PCU) updates the PMC value of each outstanding miss in each active pure miss cycle. The PCU implements the divider through a lookup table, which is fast in performance and cheap in hardware cost. Since the number of MSHR entries is limited (e.g., 64), N_x is an integer ranging from 1 to 64. Therefore, we can store all possible values of $1/N_x$ in the lookup table in advance for fast access. When a miss is served, the PMC value of the miss can be converted into a quantized integer value and stored in the tag-store entry of the corresponding cache block to guide the concurrency-aware cache management framework (Section V-F).

TABLE II: PMC of the study case.

Miss	PMC	MLP-based cost
A	0	5
C	1/3 + 1/3 + 1/3 = 1	7/3
D	1/3 + 1/3 + 1/3 + 1/2 + 1/2 = 2	7/3
E	1/3 + 1/3 + 1/3 + 1/2 + 1/2 = 2	7/3
Active pure miss cycles : 5 (cycles 10-14)		

PMC measurement in parallel multi-thread execution. In a multi-core execution, a private instance of PML is present on each core. If a core runs multiple threads, memory access from any thread contributes to the memory active cycles of that core. PMC evaluates the contribution of each cache miss to the total active pure miss cycles from the *same core*. Therefore, in a multi-threaded execution, the PMC value of each outstanding miss can be calculated on a per-core basis.

C. Revisit the Study Case with PMC Analysis

Recalling the case study in Section III-B, we re-evaluate the impact of each cache miss on performance according to the definition of PMC. When considering access concurrency and hit-miss overlapping, although access A has the highest MLP-based cost, it does not contribute to any active pure miss cycle. Therefore, the value of PMC for access A is 0. Access C has three pure miss cycles (cycle 10-12); they overlap with D's and E's pure miss cycles from cycle 10 to cycle 12. Therefore, the PMC of access C in these three cycles is 1 ($1/3 \times 3$). Access D and E have the same situation, and we take access D as an example for analysis. Access D has five pure miss cycles (cycle 10-14), which overlap with the pure miss cycles of C from cycle 10 to cycle 12 and overlap with E's pure miss cycles from cycle 10 to cycle 14. So access D has a PMC value of 2. Although access C, D, and E have the same MLP-based cost, access D and E contribute the most to active pure miss cycles and cause the most damage to performance.

In this case study, the sum of the PMC values of all cache misses is 5, which equals the number of active pure miss cycles. Table II summarizes the value of PMC and MLP-based cost for each cache miss. Compared to MLP-based cost, PMC not only considers the miss-miss overlapping but also captures the hit-miss overlapping. Therefore, PMC can better reflect the impact of each cache miss on performance.

D. Distribution of PMC

Figure 5 shows the distribution of PMC for 16 workloads from SPEC CPU2006 [43] and 2017 [6] benchmark suite. The results are measured with a single-core configuration. Details about the simulation environment are described in Section VI. LRU replacement policy is used in the LLC by default. The y-axis represents the percentage of total LLC misses, and the x-axis represents several bins for different PMC values.

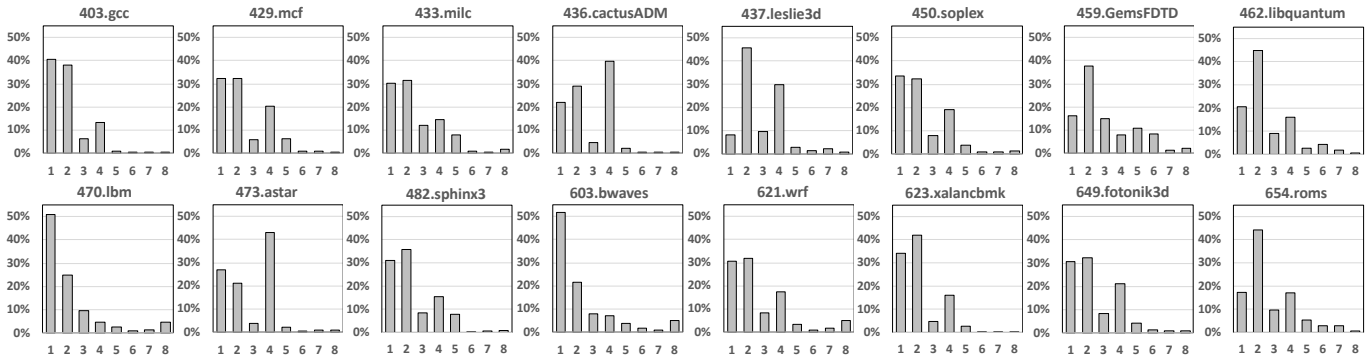

Fig. 5: Distribution of PMC. (The x-axis represents the value of PMC in cycles. 1: 0-49 cycles; 2: 50-99 cycles; 3: 100-149 cycles; 4: 150-199 cycles; 5: 200-249 cycles; 6: 250-299 cycles; 7: 300-349 cycles; 8: 350+ cycles)

TABLE III: Distribution and median of PMC_δ.

PMC_δ	403	429	433	436	437	450	459	462	470	473	482	603	621	623	649	654
[0,50)	89.40%	62.63%	64.02%	79.57%	68.19%	60.29%	57.74%	62.17%	59.99%	79.23%	57.72%	60.69%	64.77%	63.31%	50.66%	64.17%
[50,100)	3.89%	16.49%	14.52%	10.06%	18.70%	16.78%	14.95%	13.11%	16.23%	10.06%	18.18%	15.82%	15.24%	14.80%	19.85%	14.65%
[100,150)	5.56%	12.23%	12.27%	5.42%	9.33%	12.58%	11.00%	15.86%	7.22%	6.04%	12.93%	7.59%	9.09%	14.21%	16.87%	13.95%
$\geq$ 150	1.15%	8.65%	9.18%	4.96%	3.79%	10.36%	16.30%	8.87%	16.56%	4.67%	11.18%	15.90%	10.90%	7.68%	12.62%	7.23%
Median	2.87	31.00	33.00	1.00	21.00	33.33	35.13	40.00	33.44	5.03	36.00	32.44	26.00	33.50	48.75	31.25

The distribution of PMC for different benchmarks clearly shows that each cache miss has a diverse impact on performance. We can utilize PMC to grade cache misses and optimize the performance by reducing the number of pure misses with high PMC values.

E. Predictability of PMC

PC has been used successfully in predicting the reuse behavior of cache blocks [17], [20], [42], [48], [53]. We also find that the PMC value of a cache block is highly correlated with the PC of the instruction that caused the miss, which means that PMC has high predictability. Here, what predictability refers to is that the PMC value is relatively stable for all accesses for the same PC. We use PMC_δ to represent the absolute difference in PMC values between two consecutive cache misses for the same PC. Table III shows the distribution and median of PMC_δ for different workloads in a single-core configuration. These statistics come from our offline profiling of all cache misses in each workload.

For all workloads, the majority of PMC_δ values are less than 50 cycles. The median PMC_δ of each workload is also relatively low, which indicates the PMC values of a PC are almost consistent and repetitive across consecutive misses. We also observed the same trend in the multi-core configurations, as the PMC value is calculated on each core individually. The predictability of PMC provides a basis for predicting the future PMC values of the same PC. Inspired by SHiP++ [53], we design CARE as a representative use case to utilize the PMC metric. CARE is a management framework that enhances SHiP++ by enabling comprehensive concurrent access pattern analysis (more details in Section V).

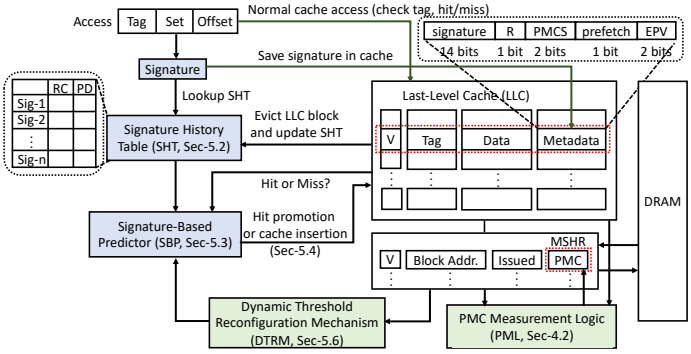

Fig. 6: Block diagram of CARE.

V. CARE: Concurrency-Aware Cache Management

We introduce CARE, an LLC management framework that considers both data locality and data concurrency. The primary goal of CARE is to improve locality while utilizing concurrency to reduce the overall performance penalty of cache misses. CARE learns the re-reference characteristics and PMC values of the cache blocks by associating each cache reference with a PC-based signature.

A. Overview

Figure 6 shows a high-level overview of CARE. Signature History Table (SHT) is designed to keep track of the observed re-reference and PMC behaviors of LLC blocks by associating them with PC signature. The purpose of Signature-Based Predictor (SBP) is to make re-reference and PMC predictions on cache insertions and cache hits.

For every new cache access, CARE first extracts the signature from the PC of the cache request. The signature of the access is used to index into the SHT. Each SHT entry tracks the

cache accesses associated with a specific PC-based signature. SHT contains two counters for each signature: Re-reference Confidence (RC) and PMC Degree (PD). RC indicates the re-reference behavior for a signature. PD reflects the cost degree of the cache miss associated with this signature. The past behaviors of the PC can be used to predict the likely re-reference and PMC characteristics of the incoming blocks. Based on SHT, SBP predicts the behaviors of each cache block. SBP then determines the insertion policy for cache misses and the promotion policy for hit accesses. CARE updates and stores the SHT on cache evictions and cache hits. To avoid the per-block overhead, Section V-G illustrates the use of set sampling [33], [48] to learn the caching behavior and update the SHT with limited overhead.

CARE uses PML to compute PMC values for all outstanding misses. The PMC values are then quantized into a 2-bit *Pure Miss Contribution States (PMCS)* by a Dynamic Threshold Reconfiguration Mechanism (DTRM) and stored in the meta-data. In order to learn the re-reference and PMC patterns of a signature, the signature (14-bit hash of PC [48], [53]), a single re-reference bit (*R*), and 2-bit *PMCS* are needed to be stored as metadata for each cache block.

B. Store and Update Access History in SHT

Metadata bits in cache blocks. The single *R* bit is used to track the re-reference behavior of each cache block. For an incoming block, the *R* bit is initially set to 0. When a miss is served, CARE quantizes the value of PMC into a 2-bit *PMCS* based on the comparison results of PMC and two thresholds PMC_low and PMC_high. If the PMC value of a miss is smaller than PMC_low, the *PMCS* of the corresponding cache block is 0. If the PMC value of a miss is larger than PMC_high, the *PMCS* of the corresponding cache block is set to 3. If the value of PMC is between the two thresholds, the corresponding *PMCS* is 1. Therefore, when an incoming block is inserted in the cache, its *PMCS* bits are set to indicate the PMC value.

SHT entry structures. The SHT has 16K entries, each containing a 3-bit RC counter and a 3-bit PD counter. CARE uses the signature of the cache access to look up the SHT. A zero RC value indicates the future blocks associated with this signature are rarely reused. A positive RC counter implies that the future blocks associated with this signature have data locality, and they are likely to receive cache hits. Similar to RC, a larger PD value means that the cache misses associated with that signature have a high probability of having large PMC values in the future.

Update SHT on hit accesses. If the cache block receives a hit, the *R* bit of the block is set to 1. If this is the first re-reference of the block, CARE increments the RC counter (in a saturated manner), which corresponds to the signature of the block in the SHT [53].

Update SHT on eviction. When a block is evicted from the cache, if the block has never been reused since it was inserted into the cache (*R* is unset), then the RC counter associated with the signature is decremented (in a saturated manner) [48], [53]. If the *PMCS* of this evicted block is 0, it means that there is a high probability that future misses caused by the same signature will hardly damage performance. Therefore, the related PD counter in the SHT is decremented (in a saturated manner). On the other hand, if the *PMCS* of this evicted block is 3, which implies that the future misses associated with the same signature are predicted to be costly, the related PD counter is incremented (in a saturated manner).

TABLE IV: Insertion and hit-promotion policy of CARE. Higher EPV value indicates higher eviction priority.

Reuse hint	Insertion policy	Hit policy
High-Reuse	EPV = 0	EPV = 0
Moderate-Reuse	if(Low-Cost) EPV = 3; else if (High-Cost) EPV = 0; else EPV = 2;	EPV = 0
Low-Reuse	EPV = 3	if(EPV $>$ 0) EPV - -;

C. Predict Access Behavior with SBP

For every cache access, SHT is indexed by a signature of the block. SBP identifies the re-reference behavior of cache blocks as High-Reuse if the associated RC counter is saturated at its maximum value. A cache block is predicted as Low-Reuse if the related RC counter is 0. All other cache accesses are classified as Moderate-Reuse.

Similarly, SBP utilizes the PD counter to predict the impact of each cache block on performance. Suppose the PD associated with the signature of the block saturates to the highest value. In that case, the cache block is predicted as High-Cost by SBP. Suppose the signature of a block has a PD value of 0. In that case, the associated cache block is predicted to be Low-Cost because the misses associated with this signature are considered less detrimental to future performance.

D. CARE Cache Management Policies

By keeping track of the reuse information, CARE can still leverage data locality to keep High-Reuse data blocks and evict Low-Reuse blocks. Additionally, CARE takes data concurrency into account for the blocks predicted to be Moderate-Reuse. CARE selectively reduces expensive misses by keeping High-Cost blocks in the cache for a longer period while giving the higher eviction priority to the Low-Cost blocks. To do so, CARE implements cache management by associating each cache block with a 2-bit *Eviction Priority Value (EPV)*.[2] The EPV counter of a cache block reflects the eviction priorities of the cache blocks. An EPV of zero implies that a cache block has the lowest eviction priority. An EPV of saturation implies that a cache block has a high eviction priority and will be evicted sooner. CARE assigns/updates the EPV counter for each block based on the prediction information from SBP. Table IV shows the updated cache policies for CARE.

Insertion policy. When inserting new blocks in the cache, unlike LRU, which inserts all cache blocks to the MRU position

[2]All advanced cache management schemes [17], [19], [42], [48], [53] have a similar counter for each cache block to indicate the eviction priority of the block. Cache management schemes implement specialized cache policies by assigning/updating the eviction priority of each block. The EPV in CARE does not introduce additional overhead.

of the "LRU chain" [19], [49], CARE dynamically learns the re-reference and PMC information of each specific signature. CARE inserts blocks with different EPV values according to the predictions provided by SBP.

Blocks that are predicted to be High-Reuse are assigned an EPV value of 0 with the lowest eviction priority. For the blocks tagged as Low-Reuse, the corresponding EPV is set to 3, and the eviction priority is the highest. Unlike other cache management frameworks [14], [53] that insert all predicted Moderate-Reuse blocks into a certain position in the cache, CARE determines the EPV value of each Moderate-Reuse block based on the PMC prediction. The EPV value of the Low-Cost blocks is assigned as 3, and the EPV value of the High-Cost blocks is assigned as 0. If a Moderate-Reuse block is neither predicted as High-Cost nor Low-Cost, its EPV value will be set to 2.

Hit-promotion policy. The EPV of a cache block will have an opportunity to be updated when the block is hit. The primary purpose of the hit-promotion policy is to preserve blocks with good locality for a long time. For a cache block that is predicted to be High-Reuse or Moderate-Reuse, the EPV drops to 0 when the block is hit to protect it from thrashing. The likelihood of further reuse is quite limited for an LLC hit to a block classified as Low-Reuse. Therefore, CARE gradually decrements its EPV value on every hit.

Victim selection. On a cache miss, CARE selects a victim block whose EPV is 3. If there are multiple candidates, CARE randomly selects one victim block from the candidates. Another solution is to consider the recency information and evict the least recent block. Through testing, there is no discernible performance difference between the two solutions. However, recording the recency information for each cache block requires a huge hardware cost. Therefore, CARE selects the candidate randomly. If there is no block with an EPV of 3, the EPV of all blocks in the cache set will be incremented and the search will be repeated until the victim block is found. By increasing the EPV value, the dead blocks can be evicted eventually.

Writeback-aware. Writeback requests are treated as non-demand background cache requests. Such requests are rarely re-referenced [53]. In order to not compete with valuable cache resources, all writebacks are inserted in the cache with an EPV of 3. The hit-promotion policy also does not apply to the writebacks.

E. Collaboration with Prefetching

PMC measurement with prefetching. We extend the definition of pure miss when there is a prefetcher in the system. No matter whether a miss is a demand miss or a prefetch miss, it is a pure miss if it has at least one miss-access cycle that does not overlap with any hit access from the *same core*. Then, we are able to calculate PMC as usual when a prefetcher exists.

CARE with prefetching. In the presence of prefetching, the caching behavior of demand accesses and prefetch accesses are completely different [18], [49], [53]. Therefore, it would be beneficial to independently predict the caching behavior of demand accesses and prefetch accesses. Inspired by SHiP++ [53], we distinguish prefetch accesses from demand accesses by appending 1-bit *prefetch* into the signature. As a result, CARE independently learns the caching behavior of load instructions that result in both demand and prefetch accesses.

In addition, CARE applies different hit-promotion policies for prefetch and demand requests on cache hits. We find a prefetched block is often only accessed once by its demand request. So, CARE sets the EPV to 3 when a prefetched block is re-referenced by a demand request. CARE updates the EPV to 0 to keep it in the cache for longer if subsequent demands or prefetch requests further access the prefetched blocked. CARE does not update the EPV of a prefetched block if it is subsequently re-referenced only by prefetch requests.

F. Dynamic Threshold Reconfiguration Mechanism

To improve the robustness of CARE, we propose a dynamic threshold reconfiguration (DTRM) scheme that quantizes PMC value into PMCS to suit different workloads. With the help of DTRM, CARE dynamically adapts two thresholds PMC_low and PMC_high to achieve the purpose of increasing adaptability. The initial PMC_high and PMC_low are set to 350 cycles and 50 cycles, respectively.

PMC_high is used to distinguish the cache block that hurts the performance the most. On a cache miss, if its PMC is larger than PMC_high, we consider the corresponding block to be a costly miss and set its PMCS to 3. A 32-bit counter TCM counts the total number of costly misses during the application execution time.

At the end of each period (16K misses, half the number of blocks in the LLC for single-core configuration), PMC_low and PMC_high are updated based on the number of costly misses found during the period. If the number of costly misses is smaller than $0.5\% \times 16K$, PMC_low and PMC_high are decreased by 10 and 70 cycles, respectively. On the other hand, if the number of costly misses is larger than $5\% \times 16K$, PMC_low and PMC_high are increased by 10 and 70 cycles, respectively. We choose the values empirically based on the results of a large number of simulations.

At the end of each period, the update of PMC_low and PMC_high drives the update of the quantization scheme between PMC and PMCS. The updated thresholds and quantization scheme are then used throughout the next period. DTRM can perfectly adapt to the mechanism of CARE. By cooperating with DTRM, the robustness and resiliency of CARE are further enhanced without any pre-processing or training overhead.

G. Hardware Cost and Complexity of CARE

We analyze the hardware cost of CARE with 2MB LLC and 64B block size as an example setting. CARE requires monitoring PMC for each cache miss during the execution time. We have introduced the measurement and implementation for tracking PMC in Section IV-B. The only needed bit per core is *NoNewAccess*. To measure the PMC value of each miss, a lookup table is used instead of a costly divider. For an LLC with 64-entry MSHR, the cost of the lookup table is 0.25KB.

TABLE V: Hardware cost of CARE (16-way 2MB LLC).

	Size	Used for
NoNewAccess(1-bit/core)	1bit	PMC
lookup table (32-bit/entry)	0.25KB	PMC
PMC(32-bit/MSHR entry)	0.25KB	PMC
PMC_low	32bit	DTRM
PMC_high	32bit	DTRM
TCM	32bit	DTRM
EPV(2-bit/block)	8KB	metadata
prefetch(1-bit/block)	4KB	metadata
signature(14-bit/sampled set)	1.75KB	metadata
R (1-bit/sampled set)	0.125KB	metadata
PMCS(2-bit/sampled set)	0.25KB	metadata
RC(3-bit/SHT entry)	6KB	SHT
PD(3-bit/SHT entry)	6KB	SHT
Total 26.64KB **(6.76KB for concurrency-aware)**		

TABLE VI: Hardware costs for different replacement frameworks (16-way 2MB LLC).

Framework	Uses PC	Concurrency-aware	Total cost
LRU	No	No	16KB
SBAR(MLP) [34]	No	Yes	28.09KB
SHiP++ [53]	Yes	No	16KB
Hawkeye [17]	Yes	No	30.94KB
Glider [42]	Yes	No	61.6KB
Mockingjay [41]	Yes	No	31.91KB
CARE	**Yes**	**Yes**	**26.64KB**

In addition, a 32-bit wide register per MSHR entry is sufficient to store the PMC value.

Each block in LLC is equipped with a 2-bit EPV for cache management and a 1-bit *prefetch* for detecting prefetch access. In order for SHT to learn the re-reference and PMC patterns of the signature, a 14-bit signature, 1-bit *R*, and 2-bit *PMCS* are needed to store for each block. CARE adopts an online set sampling method [33], [48] to reduce the storage overhead. In our study, CARE monitors the cache behaviors from 64 sampled sets, then updates the SHT. Therefore, only the blocks in the sampled set need to store these 17 bits of metadata. For a 16-way cache, the total cost of the sampled sets is 2.125KB.

The detailed hardware cost is shown in Table V. In total, the hardware overhead of CARE is around 26.64KB, which is only 1.3% of the capacity of a 2MB LLC. This cost scales linearly with the LLC capacity. We marked the additional cost, which is due to the fact that the CARE takes concurrency into account. CARE only needs 6.76KB to support concurrency awareness. Table VI compares the hardware costs of different cache management frameworks. Compared with the machine learning-based framework Glider, CARE requires much less hardware cost.

For multi-core processors, if the size of LLC is constant, we need to have a *NoNewAccess* bit for each core to detect the overlapping and measure the PMC. In addition, each block in the sampled sets needs to add a *core* tag to track which core the access comes from.

VI. Methodology

Simulated system. We evaluate CARE against prior cache management schemes using the version of ChampSim [4] used for the 1st instruction prefetching competition (IPC-1 [1]).

TABLE VII: Simulated system configurations.

Processor	1 to 16 cores, 4GHz, 8-issue width, 256-entry ROB
L1 Cache	private, split 32KB I/D-cache/core, 64B line, 8-way, 4-cycle latency, 8-entry MSHR, LRU
L2 Cache	private, 256KB/core, 64B line. 8-way, 10-cycle latency, 32-entry MSHR, LRU
L3 Cache	shared, 2MB/core, 64B line, 16-way, 20-cycle latency, 64-entry MSHR
Prefetcher	L1: next-line, L2: IP-stride
DRAM	4GB 1 channel (single-core), 8GB 2 channels (multi-core), 64-bit channel, 2400MT/s, tRP=15ns, tRCD=15ns, tCAS=12.5ns

TABLE VIII: Evaluated SPEC workloads.

Suite	Workload	MPKI	Workload	MPKI
SPEC 06	401.bzip2	1.34	403.gcc	25.55
	410.bwaves	18.35	429.mcf	26.28
	433.milc	19.00	436.cactusADM	4.99
	437.leslie3d	6.68	450.soplex	32.69
	456.hmmer	2.72	459.GemsFDTD	24.44
	462.libquantum	28.03	470.lbm	28.42
	473.astar	35.88	481.wrf	5.66
	482.sphinx3	12.96	483.xalancbmk	26.91
SPEC 17	602.gcc_s	17.77	603.bwaves_s	19.00
	605.mcf_s	55.62	607.cactuBSSN_s	3.51
	619.lbm_s	40.64	620.omnetpp_s	9.21
	621.wrf_s	19.22	623.xalancbmk_s	24.26
	625.x264_s	1.35	627.cam4_s	4.51
	628.pop2_s	2.99	649.fotonik3d_s	15.67
	654.roms_s	24.23	657.xz_s	1.58

TABLE IX: Graph datasets used in GAP workloads.

Dataset	Vertices	Edges	Description
orkut (or)	3.1M	117.2M	Social network
twitter (tw)	61.6M	1468.4M	Social network
urand (ur)	134.2M	2147.4M	Synthetic

ChampSim is a cycle-accurate simulator which was also used for 3rd data prefetching championships (DPC-3 [3]) and the 2nd cache replacement championship (CRC-2 [2]). The details of the configuration parameters are described in Table VII. For multi-core configurations, we scale the size of the LLC in proportion to the number of cores. To evaluate the performance of CARE with prefetching, we follow the methodology of CRC-2 by applying the next-line prefetching policy at L1 and IP-stride prefetching policy at L2.

Benchmarks and workloads. We evaluate CARE using a diverse set of memory-intensive workloads spanning SPEC CPU2006 [43], SPEC CPU2017 [6], and GAP [9] benchmark suites, which have at least 1 LLC miss per kilo instructions (MPKI) in the single-core baseline system with no prefetching. Table VIII shows LLC MPKI for the 30 evaluated SPEC workloads in our study. For SPEC workloads, the traces are collected with SimPoint [32] and are provided by DPC-3 [3]. For GAP workloads, we select five primitive graph algorithms for evaluation, which are Betweenness Centrality (bc), Breadth First Search (bfs), Connected Components (cc), PageRank (pr), and Single Source Shortest Path (sssp). Table IX lists the graph datasets that are used in our experiments. We use the Intel Pin dynamic binary instrumentation tool [5] to collect the traces of GAP workloads. We use the region-of-interest (ROI) utility

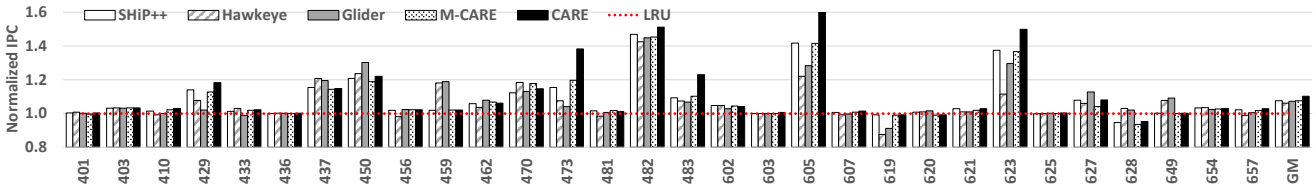

Fig. 7: Normalized IPC for all 4-core multi-copy SPEC workloads (collaboration with L1 and L2 prefetcher).

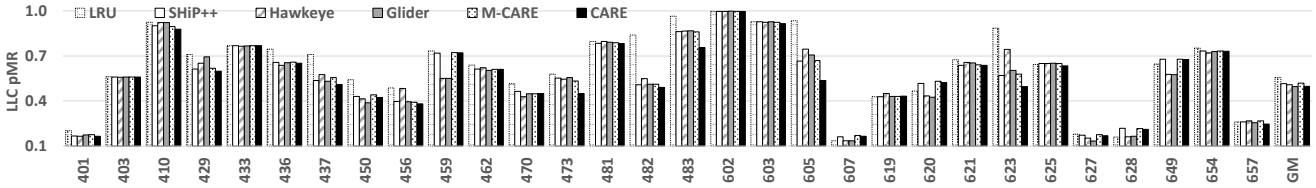

Fig. 8: LLC pMR for all 4-core multi-copy SPEC workloads (collaboration with L1 and L2 prefetcher).

from Pin to only profile the core algorithm (avoid intercepting traces when loading graph dataset). For all simulations, we warm up each core using 50M instructions from each workload, and then run simulation over the next 200M instructions.

We use both multi-copy and mixed workloads to simulate a multi-programmed system. An *n*-core multi-copy workload has *n* identical copies of a memory-intensive trace, for all the cores. Note that each trace does not start exactly at the same time in the simulator, so the runs are not synchronized. For an *n*-core mixed workload, we select *n* benchmarks randomly from the 30 memory-intensive SPEC benchmarks and run one trace in each core. We generate 100 mixed workloads in total. For each mixed workload, if a benchmark finishes early, it is replayed until each benchmark has finished running 200M instructions. Our multi-core simulation methodology is similar to the methodologies used by CRC-2 [2] and DPC-3 [3].

Compared schemes. We select LRU as the baseline for performance comparison. For multi-core configurations, we report the weighted speedup over LRU, which is commonly used to evaluate shared caches [2], [41]. We compare CARE against three state-of-the-art LLC management schemes: SHiP++ [53], Hawkeye [17], Glider [42], and Mockingjay [41]. We also extend the MLP-based cost [34] and implement a cache management framework called M-CARE for performance comparison. The workflow of M-CARE is similar to CARE. The only difference from CARE is that M-CARE does not consider PMC but uses MLP-based cost to analyze data access concurrency and guide cache management. The impact of analyzing hit-miss overlapping can be seen by comparing the performance of CARE and M-CARE.

VII. Results

A. CARE Performance Evaluation

We show the performance results of CARE over state-of-the-art schemes in 4-core configuration with prefetching in this section.

1) Multi-copy SPEC workloads: For multi-copy SPEC workloads, Figure 7 shows the normalized IPC of schemes followed by the geometric mean across all the SPEC workloads.

TABLE X: Average pMR and PMC for all 4-core multi-copy SPEC workloads (collaboration with L1 and L2 prefetcher).

	LRU	SHiP++	Hawkeye	Glider	M-CARE	CARE
pMR	0.56	0.52	0.51	0.50	0.52	0.50
PMC	114.46	97.98	99.44	101.43	97.80	95.11

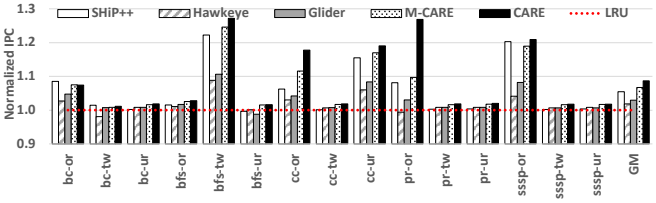

Fig. 9: Normalized IPC for all 4-core multi-copy GAP workloads (collaboration with L1 and L2 prefetcher).

All results are normalized to the baseline of LRU. In the presence of prefetching, CARE achieves a geometric mean speedup of 10.3% over the LRU on the 30 memory-intensive benchmarks, while SHiP++, Hawkeye, Glider, and M-CARE improve performance over LRU by 7.6%, 6.2%, 7.2%, and 7.5%, respectively.

Figure 7 demonstrates that since CARE considers both locality and concurrency in cache management, CARE outperforms state-of-the-art schemes. LRU, SHiP++, Hawkeye, and Glider are the locality-based schemes. Although Hawkeye and Glider each perform well for some specific workloads, their performance improvements are not consistent across different benchmarks. Hawkeye yields performance below LRU baseline for 7 workloads. Glider performs worse than the LRU baseline for 5 workloads. This observation exhibits that the performance of the locality-based schemes is limited by access patterns. M-CARE considers the miss concurrency but ignores the hit-miss overlapping. Therefore, M-CARE makes the replacement decisions at a coarse granularity. The accurate analysis of data concurrency by PMC gives the performance of CARE an advantage in comparison to M-CARE.

Figure 8 compares the LLC pure miss rate (pMR) with different schemes for the 30 memory-intensive SPEC workloads.

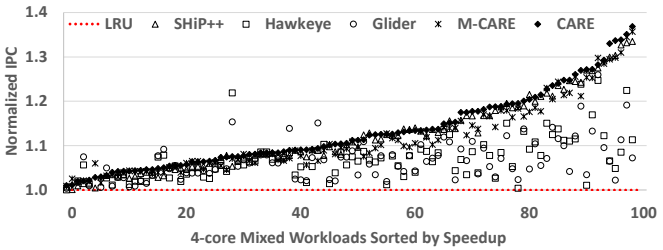

Fig. 10: Weighted Speedup for 4-core mixed workloads (collaboration with L1 and L2 prefetcher).

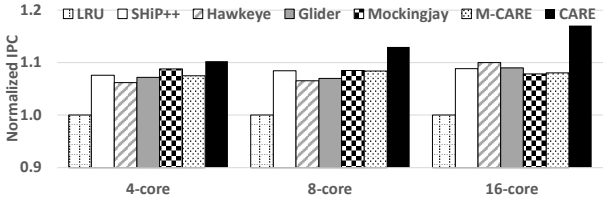

Fig. 11: Speedup for 4, 8, 16 cores (multi-copy SPEC workloads with prefetching).

On average, CARE and Glider yield the lowest average LLC pMR (0.50) than the other schemes. Table X shows the average PMC value of each LLC miss. In the CARE framework, the average PMC value is 95.11 cycles, which is less than 97.80 cycles by the second-best scheme (M-CARE in this case). The overall performance of the cache management scheme correlates well with the change in LLC pure misses and the average PMC value for all LLC misses. Compared to other schemes, CARE is not only more effective in reducing the number of LLC pure misses, but also in reducing the PMC values of LLC misses.

2) Multi-copy GAP workloads: Figure 9 shows the detailed results on the multi-copy GAP workloads. On a 4-core system with prefetching, CARE improves performance over LRU by 8.7%, while SHiP++, Hawkeye, Glider, and M-CARE improve performance by 5.4%, 1.8%, 3.0%, and 6.7%, respectively. The irregular access patterns of graph-analytic applications challenge the re-reference prediction. CARE performs better than state-of-the-art schemes, which can be attributed to the following two factors. First is its concurrency awareness, which minimizes the performance loss caused by misses. Second is the conservative nature of the hit policy. When a block incurs a hit, the eviction priority of that cache block is still reduced even if CARE predicts that the block is Low-Reuse.

3) Mixed Workloads: Figure 10 shows the normalized weighted IPC comparison between different schemes for 100 4-core mixed workloads. In the presence of prefetching, CARE offers a geometric mean speedup of 12.8% over LRU, compared with the 11.9%, 6.8%, 6.4%, and 11.4% improvements for SHiP++, Hawkeye, Glider, and M-CARE. CARE yields the best performance for 67 mixed workloads. CARE shows an advantage over other schemes in providing stable performance.

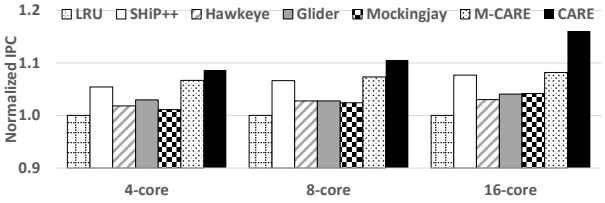

Fig. 12: Speedup for 4, 8, 16 cores (multi-copy GAP workloads with prefetching).

TABLE XI: AOCPA with increasing core numbers (collaboration with L1 and L2 prefetcher).

	4-core	8-core	16-core
AOCPA	260.87	413.03	674.89

B. Scalability Evaluation

1) Speedup with increasing concurrency: In the presence of prefetching, Figure 11 summarizes CARE performance on multi-copy SPEC workloads as we increase the number of cores. We make three major observations from Figure 11. First, the opportunity for cache management increases with multiple cores due to increased pressure on the LLC. Second, CARE outperforms every state-of-the-art cache management scheme in all configurations. Third, as the number of cores increases, the performance advantage of CARE becomes greater and greater. In the 4-core configuration, CARE achieves an improvement over LRU by 10.3% on average. As the number of cores increases, the gains are 13.0%, and 17.1%, respectively.

Similarly, Figure 12 shows the performance improvement of cache management schemes across GAP workloads in 4-core to 16-core systems. The performance improvement of CARE over other schemes increases with the number of cores. In 16-core systems, CARE outperforms LRU, SHiP++, Hawkeye, Glider, Mockingjay, and M-CARE by 16.1%, 7.8%, 12.7%, 11.6%, 11.4%, and 7.3%, respectively.

We measure *Average Overlapping Cycles Per Access (AOCPA)* on SPEC and GAP workloads to quantify the access concurrency on LLC. AOCPA is calculated on a per-core basis. A higher AOCPA means more data access overlapping in the LLC that CARE can exploit. Table XI summarizes the value of AOCPA in the presence of prefetching for multi-core configurations. Table XI shows that as the number of cores increases, the AOCPA increases significantly. This is because when the LLC receives a heavier workload, the increasing miss rate and average miss access latency lead to more miss-miss and hit-miss overlapping cycles. Therefore, in a multi-core system where the AOCPA is increasing, CARE continues outperforming other locality-only based cache management schemes and shows a huge potential for improving the performance of data-intensive applications on large-scale computing systems.

2) Performance without prefetching: Figures 13 and 14 show the scalability results without prefetching for SPEC and GAP workloads, respectively. In the absence of prefetching, the performance of CARE still scales well on SPEC and GAP

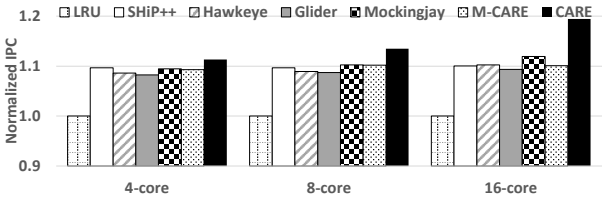

Fig. 13: Speedup for 4, 8, 16 cores (multi-copy SPEC workloads without a prefetcher).

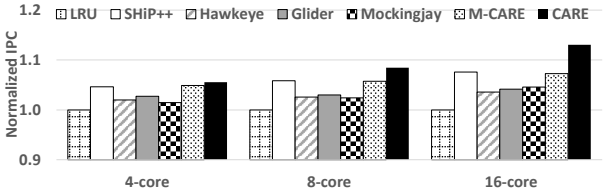

Fig. 14: Speedup for 4, 8, 16 cores (multi-copy GAP workloads without a prefetcher).

workloads with increasing concurrency. In 16-core systems, Figure 13 illustrates that CARE improves performance on SPEC workloads by 19.4%, compared with a 11.9% improvement by the second-best scheme (Mockingjay in this case). Figure 14 shows that in 16-core systems, CARE outperforms LRU, SHiP++, Hawkeye, Glider, Mockingjay, and M-CARE across GAP workloads by 13.0%, 5.1%, 9.1%, 8.5%, 8.1%, and 5.4%, respectively.

VIII. Conclusions

In this paper, we emphasize the importance of data concurrency to memory performance. We propose *Pure Miss Contribution* (PMC), a comprehensive metric used to weigh the performance cost of each cache miss. We first develop a detailed measurement mechanism for PMC. Then, we utilize PMC to build CARE, a locality and concurrency-aware, lightweight cache management framework. CARE considers locality, concurrency, and overlapping to guide cache replacement decisions. CARE is fully tested and analyzed. It outperforms state-of-the-art cache management schemes. Experimental and analysis results show CARE has a true potential for data-intensive scalable computing systems.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research is supported in part by the National Science Foundation under Grants CCF-2029014, CCF-2008907, CNS-2152497, and by the NSF supported Chameleon testbed facility.

References

[1] "1st instruction prefetching championship." https://research.ece.ncsu.edu/ipc/.
[2] "2nd cache replacement championship." https://crc2.ece.tamu.edu/.
[3] "3rd data prefetching championship." https://dpc3.compas.cs.stonybrook.edu/?final_programs.
[4] "The champsim simulator," https://github.com/ChampSim/ChampSim.
[5] "Pin-a dynamic binary instrumentation tool," https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.
[6] "Spec cpu2017 benchmark suite," http://www.spec.org/cpu2017/.
[7] A. Agarwal, K. Roy, and T. Vijaykumar, "Exploring high bandwidth pipelined cache architecture for scaled technology," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 2003, p. 10778.
[8] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-opt: Practical optimal cache replacement for graph analytics," in *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2021.
[9] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[10] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.
[11] M. Chaudhuri, "Pseudo-lifo: The foundation of a new family of replacement policies for last-level caches," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 401–412.
[12] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* IEEE, 2004, pp. 76–87.
[13] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *2012 45Th annual IEEE/ACM international symposium on microarchitecture*. IEEE, 2012, pp. 389–400.
[14] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 234–248.
[15] H. Gao and C. Wilkerson, "A dueling segmented lru replacement algorithm with adaptive bypassing," in *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.
[16] A. Glew, "Mlp yes! ilp no," *ASPLOS Wild and Crazy Idea Session*, vol. 98, 1998.
[17] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 78–89.
[18] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 110–123.
[19] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.
[20] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 436–448.
[21] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *2007 25th International Conference on Computer Design*. IEEE, 2007, pp. 245–250.
[22] M. Kharbutli and R. Sheikh, "Lacs: A locality-aware cost-sensitive cache replacement algorithm," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1975–1987, 2013.
[23] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.
[24] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 195–201.
[25] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 144–154.
[26] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, no. 12, pp. 1352–1361, 2001.
[27] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 222–233.
[28] J. Liu, P. Espina, and X.-H. Sun, "A study on modeling and optimization of memory systems," *Journal of Computer Science and Technology*, vol. 36, no. 1, pp. 71–89, 2021.

[29] Y. Liu and X.-H. Sun, "Lpm: A systematic methodology for concurrent data access pattern optimization from a matching perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2478–2493, 2019.

[30] X. Lu, R. Wang, and X.-H. Sun, "Apac: An accurate and adaptive prefetch framework with concurrent memory access analysis," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 222–229.

[31] X. Lu, R. Wang, and X.-H. Sun, "Premier: A concurrency-aware pseudo-partitioning framework for shared last-level cache," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 391–394.

[32] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[33] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[34] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 2006, pp. 167–178.

[35] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: demand-based associativity via global replacement," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 544–555.

[36] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 445–454.

[37] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, "On high-bandwidth data cache design for multi-issue processors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1997, pp. 46–56.

[38] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1990, pp. 134–142.

[39] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2012, pp. 355–366.

[40] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *Proceedings of the 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[41] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 558–572.

[42] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 413–425.

[43] C. D. Spradling, "Spec cpu2006 benchmark tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[44] X.-H. Sun and D. Wang, "Concurrent average memory access time," *Computer*, vol. 47, no. 5, pp. 74–80, 2013.

[45] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[46] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 409–422.

[47] W. A. Wong and J.-L. Baer, "Modified lru policies for improving second-level cache behavior," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, 2000, pp. 49–60.

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

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

[50] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[51] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 174–183, 2009.

[52] L. Yan, M. Zhang, R. Wang, X. Chen, X. Zou, X. Lu, Y. Han, and X.-H. Sun, "Copim: a concurrency-aware pim workload offloading architecture for graph applications," in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2021, pp. 1–6.

[53] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *Proceedings of the Cache Replacement Championship (CRC'17) held in Conjunction with the International Symposium on Computer Architecture (ISCA'17)*, 2017.