2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

CHROME: Concurrency-Aware Holistic Cache Management Framework with Online Reinforcement Learning

Xiaoyang Lu†, Hamed Najafi‡, Jason Liu‡ and Xian-He Sun†
†Illinois Institute of Technology, ‡Florida International University
xlu40@hawk.iit.edu, hnaja002@fiu.edu, liux@cis.fiu.edu, sun@iit.edu

Abstract—**Cache management is a critical aspect of computer architecture, encompassing techniques such as cache replacement, bypassing, and prefetching. Existing research has often focused on individual techniques, overlooking the potential benefits of joint optimization. Moreover, many of these approaches rely on static and intuition-driven policies, limiting their performance under complex and dynamic workloads. To address these challenges, this paper introduces CHROME, a novel concurrency-aware cache management framework. CHROME takes a holistic approach by seamlessly integrating intelligent cache replacement and bypassing with pattern-based prefetching. By leveraging online reinforcement learning, CHROME dynamically adapts cache decisions based on multiple program features and applies a reward for each decision that considers the accuracy of the action and the system-level feedback information. Our performance evaluation demonstrates that CHROME outperforms current state-of-the-art schemes, exhibiting significant improvements in cache management. Notably, CHROME achieves a remarkable performance boost of up to 13.7% over the traditional LRU method in multi-core systems with only modest overhead.**

I. Introduction

With the advancement of large-scale, data-intensive applications, optimizing the performance of modern memory systems has become crucial for achieving efficient execution. Cache hierarchy, designed to bridge the performance gap between the processor and the main memory, plays a pivotal role in memory systems [21], [25], [33], [43], [53]. With the escalating performance demands and an increasing number of on-chip cores, cache hierarchy in modern processors continues to grow both in depth and capacity [49]. Obviously, unrestricted growth of cache resources is unfeasible due to area and power budget constraints [13], [36]. Consequently, computer architects primarily rely on efficient cache management strategies.

There are three primary cache management techniques to enhance cache utilization: cache replacement, bypassing, and prefetching. Cache replacement policies (e.g., [4], [13], [21], [23], [31], [35], [41], [43], [44], [55], [58]) determine the eviction of cache blocks to accommodate new data, typically prioritizing swift eviction of blocks with large predicted reuse distance. Cache bypassing techniques (e.g., [11], [30], [36]) make decisions on whether to cache incoming blocks or have them bypass the cache, thus preventing the cache from being "polluted" with infrequently reused data. Hardware prefetchers (e.g., [6], [7], [14], [15], [33], [38], [48]) learn from complex memory access patterns and proactively fetch data most likely required by future accesses to reduce latency.

Cache replacement, bypassing, and prefetching collectively constitute the cornerstone of effective cache management, significantly contributing to the overall system performance. However, current studies often examine cache replacement, bypassing, and prefetching in isolation, overlooking the potential benefits that could arise from a joint optimization strategy. For example, demand accesses and prefetching accesses frequently exhibit distinct behaviors, necessitating a replacement policy that is cognizant of prefetching dynamics. Additionally, considering that a significant portion of cache blocks experience only a single access, an effective method capable of predicting and bypassing these blocks could lead to significantly improved performance due to reduced cache pollution. Operating in isolation, one might fail to recognize the complementary nature of these techniques, missing the opportunities for more efficient utilization of cache resources. It is advantageous to integrate cache replacement, bypassing, and prefetching as a cohesive approach rather than treating them as separate techniques.

Traditional cache management schemes are largely guided by human intuition, which is grounded in high-level assumptions about application behaviors and memory access patterns. However true, these assumptions may not always hold, leading to potentially sub-optimal policies for complex workloads and configurations. The diversity of modern workloads exacerbates the situation, making prediction and optimization of cache performance increasingly difficult. There is a need for automated techniques that can accurately predict and adapt to access patterns across various workloads. Reinforcement learning provides a promising solution, offering the potential for developing an intelligent cache management framework that learns from interactions and adapts cache decisions in response to different workloads and configurations.

In this paper, we introduce CHROME, a concurrency-aware holistic last-level cache management framework that leverages online reinforcement learning. We model cache management as a reinforcement learning problem, where the agent learns by interacting with its environment, with each action yielding a specific reward. The ultimate goal of the agent is to maximize cumulative rewards, thus driving the continuous and autonomous optimization of its policies and actions [52].

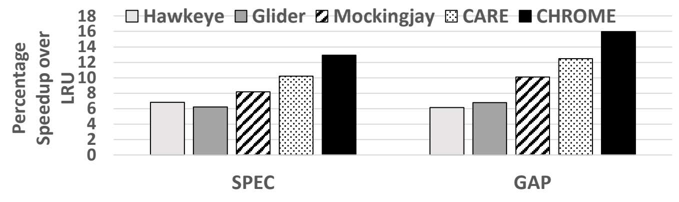

Fig. 1: Comparison of performance improvement with SOTA cache management schemes over LRU on a 16-core system (using homogeneous workload mixes).

CHROME has several unique features:

1) CHROME is a holistic cache management framework that integrates cache bypassing and replacement policies with pattern-based prefetching. The framework provides the opportunity for joint optimization, leveraging the complementary strengths of cache replacement, bypassing, and prefetching to minimize interference between separate operations that could lead to competing decisions.
2) CHROME operates as a reinforcement learning agent, conducting online learning based on predefined rewards and performance objectives. This unique design eliminates the necessity for offline training, thus avoiding potential constraints imposed by fixed policies that struggle to adapt to dynamic workloads and configurations.
3) CHROME observes multiple program features, including control-flow and data-access characteristics, and represents the last-level cache (LLC) access as a state vector. By utilizing multiple program features, CHROME significantly enhances its capacity to accurately capture the intricate memory access behavior of applications, ultimately boosting the efficacy of its learning process.
4) CHROME defines a reward for each action that considers the system-level feedback information. Both data locality and data access concurrency of the system are presented to CHROME to better evaluate the impact of its decisions through reinforcement learning.
5) CHROME is lightweight requiring only a modest hardware implementation overhead (with the smallest storage overhead among all state-of-the-art cache management schemes we consider), thus clearing the hurdles for its practical application.

We evaluate CHROME against four state-of-the-art (SOTA) cache management schemes, including Hawkeye [21], Glider [44], Mockingjay [43], and CARE [35], across a variety of memory-intensive workloads. Figure 1 shows that CHROME outperforms these schemes when running multi-programmed workloads on a 16-core system. Overall, CHROME consistently improves performance in the system with prefetching across various SPEC CPU [46], [47] and GAP [3] workloads. On average, CHROME improves performance by 13.7% over the classic Least Recently Used (LRU) baseline. CHROME outperforms Hawkeye, Glider, Mockingjay, and CARE by 6.6%, 6.9%, 4.6%, and 2.6%, respectively.

The rest of the paper is organized as follows. We present the background in Section II. In Section III, we identify the major issues in current cache management schemes, which motivated the design of CHROME. We formulate the cache management as a reinforcement learning problem in Section IV, and present the design of CHROME in Section V. CHROME is a self-optimizing framework that can dynamically and autonomously make cache management decisions at the LLC, utilizing multiple program features and concurrency-aware system-level information. Sections VI and VII present an extensive performance evaluation study to demonstrate the adaptability and scalability of CHROME. We discuss the related works in Section VIII. Finally, we conclude the paper in Section IX.

II. Background

A. Cache Management Schemes

Efficient use of the LLC can mitigate the ever-widening performance gap between CPUs and memory. To ensure that the cache retains useful blocks and minimizes cache misses, recent cache management schemes [21], [42]–[44], [53], [55] draw insights from historical access behavior to predict the future behavior of incoming blocks, optimizing cache management and improving overall system performance.

Hawkeye [21] emulates and learns from Belady's OPT policy [5] based on an extensive history of cache accesses to predict the reuse characteristics of future accesses. It formulates reuse prediction as a binary classification problem and employs a PC-based predictor to determine whether an incoming line will be cache-friendly or cache-averse. When a cache miss occurs, any block that is predicted to be cache-averse is selected for eviction. Glider [44] applies an offline attention-based long short-term memory (LSTM) model to the cache replacement problem. Data derived from Belady's OPT policy are used to train the LSTM model offline, which leads to a simpler online model based on a support vector machine. Following in the footsteps of Hawkeye, Mockingjay [43] introduces holistic thinking to guide cache replacement and bypassing decisions, particularly in the presence of prefetching. Rather than relying on binary predictions, Mockingjay effectively emulates Belady's OPT policy by basing its decisions on multi-class reuse prediction. It estimates the reuse distances for each program counter (PC) at a fine granularity, leading to the quick eviction or bypassing of blocks predicted to be reused furthest. CARE [35] stands out from other reuse distance prediction-based schemes by considering both data locality and concurrency in its cache insertion and hit-promotion decisions. It not only aims to minimize cache misses, but also works effectively to eliminate the costly ones. In scalable systems with large numbers of concurrent memory accesses, CARE demonstrates good scalability.

CHROME drew inspiration from these schemes, yet it differs significantly as an integrated approach encompassing cache replacement, bypassing, and prefetching, and as an online reinforcement learning algorithm to cope with dynamic workloads and varying system configurations.

B. Reinforcement Learning for Cache Management

Reinforcement learning (RL) [40], [52] is a machine learning technique that enables an agent to autonomously learn to maximize the cumulative reward received over its lifetime through feedback from actions and experiences in an interactive environment. The agent-environment interaction at timestep t can be expressed as a tuple (S_t, A_t, R_{t+1}), where the agent observes the state of the environment S_t and selects an action A_t, after which the environment transitions its state from S_t to S_{t+1} and provides a numerical reward R_{t+1} for the agent. The goal of an agent is to find an optimal policy that can maximize the total cumulative reward from the environment in the long term. An agent must consider the long-term impact of each action rather than focusing solely on the immediate reward. The expected value of the cumulative reward that is obtained when executing an action A in a given state S is defined as the Q-value of the state-action pair $Q(S, A)$ [54].

SARSA [40] is an on-policy algorithm for learning a Markov decision process policy, where an agent interacts with the environment and updates the Q-value depending on the current state S_t, current action A_t, reward obtained R_{t+1}, next state S_{t+1}, and next action A_{t+1}:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)].$$

The learning rate α determines the rate at which Q values are updated. The discount factor γ determines how future rewards are weighed against immediate rewards. When γ approaches 0, the agent becomes more “opportunistic” and chooses actions that favor the immediate rewards from the environment. As γ increases, the agent becomes more “far-sighted” and strives for higher rewards in the long term.

The RL framework has been successfully applied to system optimization in areas such as memory scheduling [20], data prefetching [6], [61], data placement in storage systems [45], and HPC job scheduling [59], [60]. In this work, we posit RL is suitable for holistic cache management for four main reasons: *1) Adaptive online learning.* The RL agent learns online and optimizes its policy through interaction with the environment. This continuous learning process equips the agent with the ability to adapt to various configurations and workloads with different access patterns. *2) Multiple features.* Accurate prediction of the memory access pattern of various workloads is essential for effective cache management. The use of multiple program features can improve prediction accuracy, resulting in better performance [6], [25]. The state in RL can be defined as a multi-dimensional vector of program features. *3) Rewards from environment.* The RL framework learns autonomously based on feedback from the environment. A well-designed reward structure for cache management tasks allows RL agents to take appropriate actions by considering both data locality and concurrency. *4) Acceptable overhead.* While machine learning techniques have been applied to cache management [31], [44], [53], they often involve significant overhead, including the cost of training, model space requirements, and computational costs. An RL-based framework eliminates offline training and requires a relatively small model. The reward functions are simple to compute, and the Q-values for state-action pairs can be stored in a lookup table using only moderate computational resources for inference.

C. Concurrent Memory Access Model

In modern processors, concurrency is widely adopted to mitigate the impact of long latency in accessing off-chip main memory [8]. Modern high-performance processors with advanced cache techniques, such as multi-port [62], pipelined [1], and non-blocking [28], improve the throughput by enabling multiple data accesses to overlap in the same cycles, resulting in increased concurrency of data accesses.

Concurrent Average Memory Access Time (C-AMAT) [50] is a memory performance model that quantifies the average real time spent for each memory access. C-AMAT quantifies the combined impact of locality and concurrency of memory accesses, while taking into account the overlapping of data accesses. C-AMAT can be calculated as the memory active cycles divided by the number of memory accesses. The memory active cycles are the number of cycles with active memory accesses, excluding cycles without memory references. These cycles are defined carefully to account for overlap—only one cycle is counted when multiple memory accesses occur concurrently in the same cycle at a memory layer [32], [34], [35], [50], [57]. In a multi-core system, C-AMAT tracks the memory active cycles and memory accesses from each core. Both memory active cycles and the number of memory accesses can be directly measured by Intel Performance Monitoring Units (PMUs), which have already been integrated into modern processors [17], [51]. As such, the C-AMAT value can be monitored without imposing additional overhead. The C-AMAT model can be generally applied to any level of the memory hierarchy [37], providing an accurate performance analysis of memory systems where concurrent memory accesses are prevalent. In this work, we employ the C-AMAT model to provide accurate system-level feedback information, which turns out to be important for evaluating the cache management decisions for reinforcement learning.

III. Motivation

We identify two major issues in the current cache management schemes: the lack of a holistic view of cache management and the lack of adaptability for handling complex and diverse workloads and system configurations.

A. Lack of Holistic View

Although state-of-the-art cache management schemes (such as Hawkeye [21], Glider [44], and CARE [35]) strive to keep cache blocks with better data locality in the LLC and minimize thrashing caused by blocks with large reuse distances, they focus only on a specific aspect and thereby miss the opportunity to integrate cache bypassing in the presence of prefetching.

As an example, Figure 2(a) depicts the percentage of LLC evicted cache blocks that are not reused before eviction. The target system is a 4-core system. We use the next-line prefetcher at L1, stride prefetcher at L2 [14], [15], and Glider

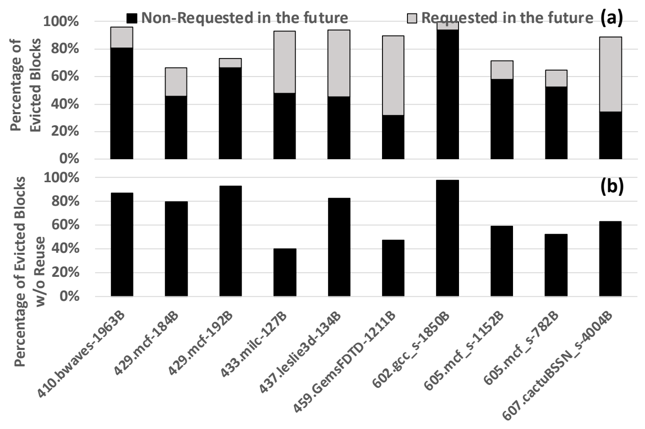

Fig. 2: Inspecting Unreused Blocks in LLC with Glider [44]: (a) the fraction of blocks not reused before eviction, and (b) the fraction of unused prefetched blocks among all blocks that are not reused before eviction.

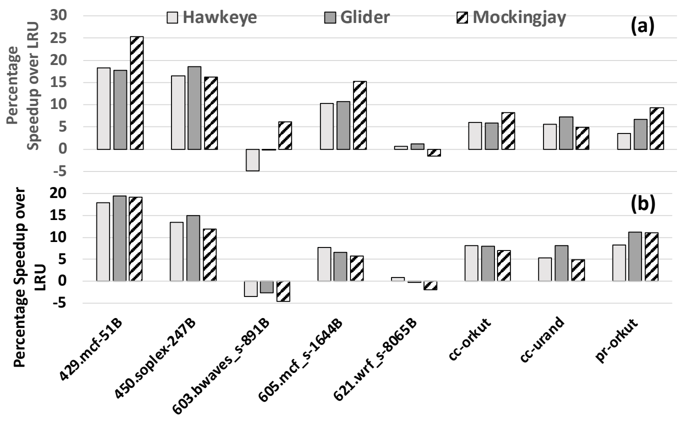

Fig. 3: Comparing speedup over LRU on a 4-core system between: (a) using next-line prefetcher at L1 and stride prefetcher at L2, and (b) using stride prefetcher at L1 and streamer prefetcher at L2.

[44] as the LLC management scheme (see Section VI for details). On average, 83.7% of evicted blocks in a 12MB shared LLC are not reused before eviction. The unused evicted blocks consist of those requested again in the future (28.0%, depicted in gray on top) and those never requested again (55.7%, in black at the bottom). Figure 2(b) shows that, on average, 70.0% of the blocks not reused before eviction come from prefetching.

Figure 2 underscores the need for a holistic cache management scheme for modern computing systems. On the one hand, bypassing is effective for blocks accessed only once. On the other hand, cache management needs to be aware of prefetching, since retaining unnecessary prefetched blocks may lead to the eviction of vital data. These considerations call for a cache management framework that seamlessly integrates replacement, bypassing, and prefetching. In response, we introduce CHROME for a holistic approach to cache management, which adeptly determines whether one should cache or bypass incoming blocks, and selectively keep the prefetched blocks, all at the same time.

B. Lack of Adaptability

Mockingjay [43] integrates cache replacement and bypassing, and designs distinct policies for demand accesses and prefetch accesses. However, the policies are statically designed based on fixed assumptions that may not be reflective of the dynamic nature of the workloads and thus can be ineffective across a broad spectrum of workload demands and diverse system configurations.

Figure 3 shows the performance of Hawkeye [21], Glider [44], and Mockingjay [43] under two different multi-level hardware data prefetching schemes using eight representative workloads. When employing a next-line prefetcher at L1 and a stride prefetcher at L2, Figure 3(a) shows that, although Mockingjay integrates replacement, bypassing, and prefetching in its design, it exhibits better performance with some workloads, but not others (`soplex`, `wrf`, and `cc-urand`). The unstable performance of Mockingjay across different workloads can be attributed to the limitations of statically-designed policies due to their lack of adaptability. Figure 3(b) depicts the performance comparison of the same schemes under the same workloads, but utilizing a stride prefetcher at L1 and a streamer prefetcher at L2. Notably, Mockingjay underperforms Glider across all workloads.

Figure 3 highlights the pivotal role of adaptive cache management. First, the inconsistent performance of static cache management policies across varied workloads accentuates the need for adaptability. Second, the limitations of static, intuition-driven policies are apparent when dealing with diverse system configurations. An adaptive framework shall be able to handle diverse workloads and system configurations. CHROME is an online reinforcement learning-based cache management framework, designed to ensure adaptability and achieve robust and consistent performance for a wide range of scenarios.

IV. CHROME: RL Formulation

We formulate cache management as an RL problem. In particular, we design CHROME as an RL agent, with the processor and the memory system acting as the environment. The primary objective of CHROME is to enhance the overall cache performance of the workloads running in a particular system configuration and adjust cache management decisions pertaining to the dynamic nature of the workloads. Each time step corresponds to a new cache access, during which CHROME observes multiple program features of the current demand or prefetch access, the processor, and the memory system, formulates them as a state, and subsequently makes the decision on an action to bypass, replace, or promote a cache block. In any case, the action works on the environment. CHROME then receives a reward that considers the accuracy of the action and system-level feedback information. This process repeats itself continuously, enabling CHROME to learn and adapt to the changing environment.

TABLE I: Program features considered for this study.

Control-flow	Data-access	Combination
PC	Memory address	PC + delta
Sequence of last 4 PCs	Memory address delta	PC + page number
	Sequence of last 4 deltas	PC + page offset
	Page number	
	Page offset	

A. State

For each cache access, CHROME utilizes observed program features to select the most suitable action. Prior research has highlighted program features that are closely correlated to the reuse distance of blocks for cache optimization [6], [25], [30], [43], [55]. In our study, we incorporate different types of program features to characterize various workloads and memory behaviors. This approach enables CHROME to gain a more comprehensive understanding of diverse workloads and memory behavior, offering multiple perspectives for more effective learning.

We define the state as a multi-dimensional vector of program features. Each program feature can either be a control-flow feature (e.g., PC), a data-access feature (e.g., memory address, page number, page offset), or a combination of these features (e.g., bits from the PC and memory address can be composed by hashing or concatenation). Table I lists the possible program features. It is important to note that, although observing a large number of features could theoretically improve learning, it can also increase overhead and pose practical challenges. Therefore, we consider the trade-off between performance and overhead, and apply feature selection [27] to determine which features to be included in the state vector. In this study, we define the state as a 2-dimensional vector: $S_t = (\mathrm{PC}_t, \mathrm{PN}_t)$. PC_t is the *signature* of program counter of the current memory instruction. PC has been used extensively in earlier studies to describe program behavior and has proven effective for cache block reuse prediction [21], [25], [30], [43], [44], [55]. We combine the PC and the hit/miss information into a hashed PC signature, which allows CHROME to distinguish between hit and miss accesses initiated at the same PC. PN_t is the physical page number of the current memory access. Due to the similarity of access patterns at memory pages, the page number represents the data-access feature that can complement the control-flow feature (PC) and provide additional insight into the program behavior.

Prefetching is an important technique for modern high-performance processors [33], [48]. In the presence of prefetching, the behavior of demand and prefetch accesses can be quite different. For instance, at a particular phase of a workload, the reuse distance of the demand accesses might be large, and therefore the corresponding data blocks should be bypassed. However, the prediction from the prefetcher might be accurate, and therefore the prefetched blocks should be inserted into the cache in time [30], [56]. Inspired by [35], [58], to distinguish between demand accesses generated and prefetch accesses triggered by the same load instruction, we hash the PC signature of each access with a *is_prefetch* bit. As a result, CHROME can learn the caching behavior of demand accesses and prefetch accesses independently.

In a multi-core system, especially when the cores are executing different applications simultaneously, accesses generated by the cores are mixed in the LLC, making it challenging to observe the access behavior of each core accurately. Therefore, in order to identify the accesses from different cores, we further hash the PC signature with the core identifier to produce a composite signature as a program feature 'PC+core' that CHROME needs to observe in multi-core systems.

B. Action

For each access, given a certain state, CHROME selects an action to guide cache management decisions. Upon a cache miss, CHROME determines whether the incoming block should bypass the LLC or be placed in the cache, in which case it is assigned with one of three possible *Eviction Priority Values* (EPVs)[1]. The EPV of a cache block designates its eviction priority; a lower EPV implies a lower eviction priority, while a higher EPV indicates that the cache block is prioritized for eviction. In the case of a cache hit, CHROME updates the EPV of the corresponding block, selecting one of the three possible levels according to the current policy.

C. Reward

In RL, the decisions of the agent are reward-driven: the goal of the agent is to obtain the maximum cumulative reward. For CHROME, the reward structure needs to reflect an evaluation of the accuracy of each action (in terms of achieving a desirable outcome) by taking into account system-level feedback.

C-AMAT [50] can quantitatively measure the combined impact of memory access locality and concurrency, taking all types of memory access overlapping into account. We employ the C-AMAT model in this study to provide precise system-level feedback information to CHROME. The purpose of having cache hierarchy is to provide faster access to memory resources, thus avoiding time-consuming accesses to off-chip main memory. However, not all workloads can benefit from this. Particularly in multi-core systems, workloads running on different cores contend for the shared LLC, causing interference with one another. Assuming workloads are bound to cores, during a runtime period (100K cycles in this study), if the concurrent average access time to LLC from $core_i$ is greater than the average latency of main memory, that is, $\text{C-AMAT}_\text{i}(\text{LLC}) > T_\text{mem}$, it indicates that there is little performance benefit for $core_i$ to cache the blocks at LLC during this period, and we call this situation *LLC-obstruction* for $core_i$. In this study, we employ the C-AMAT model to monitor the behavior of LLC-obstruction cores during runtime, utilizing it as system feedback information to structure our rewards.

[1]Advanced cache management schemes (such as [21], [23], [35], [43], [44], [55]) utilize a similar counter for each cache block, which serves to indicate the eviction priority of that block. These schemes enforce specialized cache policies by periodically assigning or updating the eviction priority of each block. Adopting EPVs in CHROME does not result in additional overhead.

We define four different rewards: R_{AC}, R_{IN}, $R_{\text{AC-NR}}$, and $R_{\text{IN-NR}}$. *1)* The reward R_{AC} is assigned to an action when its corresponding address is requested by either a demand or prefetch access, and its corresponding block is in the cache (cache hit). We further denote R^{D}_{AC} for demand access and R^{P}_{AC} for prefetch access. *2)* The reward R_{IN} is assigned to an action when the corresponding address is requested by either a demand or prefetch access, and its corresponding block has been evicted or bypassed (cache miss). Similarly we further denote R^{D}_{IN} for demand access and R^{P}_{IN} for prefetch access. *3)* The reward $R_{\text{AC-NR}}$ is granted to an action, which can be a bypassing action on a cache miss or assigning the block with the highest EPV on a cache hit, when its corresponding address is not requested by any demand or prefetch access within a temporal window. We further differentiate this reward according to the system-level feedback: $R^{\text{OB}}_{\text{AC-NR}}$, if the corresponding core is LLC-obstructed, and $R^{\text{NOB}}_{\text{AC-NR}}$, otherwise. *4)* The reward $R_{\text{IN-NR}}$ is granted to an action, which can be a non-bypassing action on a cache miss or assigning the block with anything but the highest EPV on a cache hit, when its corresponding address is not requested by a demand or prefetch access within a temporal window. Similarly, we further differentiate this reward according to the system-level feedback: $R^{\text{OB}}_{\text{IN-NR}}$, if the corresponding core is LLC-obstructed, and $R^{\text{NOB}}_{\text{IN-NR}}$, if not.

The above reward structure in CHROME is designed to achieve four key objectives. First, it provides positive rewards for accurate actions leading to cache hits and assigns negative rewards (penalties) for inaccurate actions leading to cache misses. Doing so will incentivize CHROME to make more accurate decisions to reduce cache misses. Second, it differentiates the rewards based on whether the current request is triggered by a demand or prefetch access. Such differentiation will encourage CHROME to prioritize retaining blocks likely to be requested next by demand accesses over those possibly requested by prefetch accesses. Third, if a corresponding address is predicted not to be requested by any demand or prefetch access in a temporal window, we provide rewards to incentivize the agent to bypass on a cache miss or to assign the highest EPV to the block on a cache hit. Fourth, if a corresponding address is not requested by any demand or prefetch access within a temporal window, CHROME grants either a larger positive reward or a larger penalty to the action depending on whether the corresponding core is identified as an LLC-obstructed core. This approach promotes actions that can alleviate LLC obstruction, thereby enhancing overall system performance. To achieve the above objectives, we select the reward values empirically. Table II shows the specific reward values used for this study.

V. CHROME: RL Design

A. *Overview of the RL Framework*

CHROME consists of two separate tasks, an RL decision task and an RL training task, which can run in parallel. In order to implement these two tasks, we utilize two hardware structures: the Q-Table and the Evaluation Queue (EQ). Q-Table is designed to track the Q-values of all observed state-action pairs. EQ functions as a first-in-first-out queue with a fixed capacity. Its primary role is to record the actions of CHROME within a temporal window, thereby facilitating the evaluation and reward of each action. Each EQ entry records five pieces of information: the state vector, the action executed by CHROME, whether the action was triggered by a hit or a miss, the memory address of the requested cache block, and the assigned reward. Figure 4 presents a high-level overview of the CHROME framework.

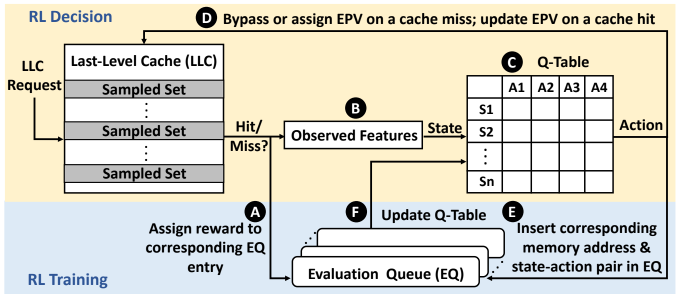

Fig. 4: Overview of CHROME.

RL decision task. For each LLC access, CHROME makes a decision. CHROME observes the program features (PC and page number) of the current access and incorporates them into a state vector (B). CHROME then searches the Q-Table to obtain the Q-values for all possible state-action pairs. Each pair is composed of the given state and one of the potential actions (C). CHROME selects the action with the maximum Q-value in the current state or chooses a random action with a given probability (exploration), then executes the action in the cache hierarchy (D). On a cache miss, the incoming cache block will either be inserted into the LLC with an assigned EPV or be bypassed. On a cache hit, the EPV of the corresponding block is updated according to the chosen action.

RL training task. Since the effectiveness of each action performed by CHROME cannot be immediately evaluated, CHROME employs EQ to record recent actions, subsequently assigning rewards to each. CHROME records the recently executed action, a 1-bit indicator denoting whether the action was triggered by a hit or miss, the corresponding state vector, and the corresponding memory address as a new entry in the EQ (E).

For each new LLC request, if the request address matches the address stored in an EQ entry (indicating that CHROME has previously executed an action for this address and this address is now being requested again within a temporal window), CHROME assigns a reward to the corresponding EQ entry (A). This assignment is based on whether the current request results in a cache hit (signifying that the action associated with the EQ entry has effectively led to a cache hit) or a miss (indicating that the action associated with the EQ entry is not sufficiently accurate).

The size of the EQ is finite. If an evicted entry never receives a reward (indicating that the corresponding address is not requested within a temporal window), CHROME assigns a

Algorithm 1 Reinforcement learning-based cache management algorithm

```
 1: procedure CHROME (addr)                                  ▷ Called for every LLC request
 2:   if sampled_set (addr) then
 3:     entry ← search_EQ (addr)      ▷ When a sampled set is accessed, search the corresponding EQ with the requested address
 4:       if entry is valid and has_reward (entry) == false then
 5:         if addr hits in a sampled set then
 6:           entry.reward ← $R_{AC}^{D}$ or $R_{AC}^{P}$    ▷ If the request hits the sampled set, assign reward $R_{AC}^{D}$ or $R_{AC}^{P}$
 7:         else
 8:           entry.reward ← $R_{IN}^{D}$ or $R_{IN}^{P}$    ▷ If the request misses the sampled set, assign reward $R_{IN}^{D}$ or $R_{IN}^{P}$
 9:   state ← get_state ( )                                  ▷ Extract the state vector from the current request
10:   if addr misses in LLC then
11:     if rand ( ) < $\epsilon$ then
12:       action ← random_action ( )     ▷ Perform exploration with a low probability $\epsilon$, select a legal random action
13:     else
14:       action ← $argmax_a Q$ (state, a)   ▷ Select the action (insert the incoming block in LLC with an EPV or bypass) with the highest Q-value
15:   else
16:     if rand ( ) < $\epsilon$ then
17:       action ← random_action ( )     ▷ Perform exploration with a low probability $\epsilon$, select a legal random action
18:     else
19:       action ← $argmax_a Q$ (state, a)   ▷ Select the action (update the EPV of the corresponding block in LLC) with the highest Q-value
20:   execute action
21:   if sampled_set (addr) then                             ▷ If the action occurs on a sampled set
22:     new_entry ← create_EQ_entry (addr, state, action, trigger)   ▷ Create a new EQ entry
23:     evict_entry ← insert_EQ (new_entry)                  ▷ Insert the entry to EQ and get the evicted EQ entry
24:     if has_reward (evict_entry) == false then
25:       if evict_entry.trigger == miss then    ▷ If the evicted entry does not have a reward, and triggered by miss
26:         if evict_entry.action == BYPASS then
27:           evict_entry.reward ← $R_{AC-NR}^{OB}$ or $R_{AC-NR}^{NOB}$   ▷ In case of bypassing action, assign reward $R_{AC-NR}^{OB}$ or $R_{AC-NR}^{NOB}$
28:         else
29:           evict_entry.reward ← $R_{IN-NR}^{OB}$ or $R_{IN-NR}^{NOB}$   ▷ Otherwise assign reward $R_{IN-NR}^{OB}$ or $R_{IN-NR}^{NOB}$
30:       else                                 ▷ If the evicted entry does not have a reward, and triggered by hit
31:         if evict_entry.action == $EPV_H$ then
32:           evict_entry.reward ← $R_{AC-NR}^{OB}$ or $R_{AC-NR}^{NOB}$   ▷ In case of assigning $EPV_H$ action, assign reward $R_{AC-NR}^{OB}$ or $R_{AC-NR}^{NOB}$
33:         else
34:           evict_entry.reward ← $R_{IN-NR}^{OB}$ or $R_{IN-NR}^{NOB}$   ▷ Otherwise assign reward $R_{IN-NR}^{OB}$ or $R_{IN-NR}^{NOB}$
35:     $R$ ← evict_entry.reward                             ▷ Get the reward stored in the evicted EQ entry
36:     $S_1$ ← evict_entry.state; $A_1$ ← evict_entry.action   ▷ Get the state and action from the evicted EQ entry
37:     $S_2$ ← EQ.head.state; $A_2$ ← EQ.head.action        ▷ Get the state and action from the entry at the head of the EQ
38:     $Q(S_1, A_1) \leftarrow Q(S_1, A_1) + \alpha [R + \gamma Q(S_2, A_2) - Q(S_1, A_1)]$   ▷ Update Q-Table, based on SARSA
```

reward based on the corresponding action recorded in that entry, the trigger (hit or miss) of the action, and system-level feedback information. Finally, the state vector, action, and reward of the evicted entry are utilized to update the corresponding Q-value in the Q-Table (F).

B. RL-based Cache Management Algorithm

Algorithm 1 details how CHROME makes decisions and performs online learning. Initially, all Q-values in the Q-Table are set optimistically to the highest possible Q-value ($\frac{1}{1-\gamma}$), encouraging CHROME to explore the environment early in the execution [52]. CHROME is trained by observing the accesses to several sampled cache sets. Prior studies show that memory access patterns are consistent across cache sets [25], [26], [35], [55]. It is thus sufficient to train CHROME by observing only accesses to a small number of sets (more details in Section V-D).

For an LLC request with address *addr*, if it belongs to a sampled set, CHROME searches EQ for *addr* (line 3). If a match is found in the EQ and it does not have a reward, CHROME assigns a reward to the corresponding EQ entry. This reward is determined based on whether the corresponding action results in a cache hit or a miss, and whether the LLC request is triggered by demand or prefetch (lines 4-8).

For every LLC access, CHROME extracts the state vector from the observed program features (line 9). CHROME either selects an action randomly to explore the environment (line 12 for cache miss and line 17 for cache hit) or refers to the Q-Table based on the given state vector and selects the action with the highest Q-value (line 14 for cache miss and line 19 for cache hit). After performing the selected action (line 20), if it is carried out on a sampled set, CHROME creates a new EQ entry with the state vector, the selected action, the corresponding address, and the trigger (lines 21-22). CHROME then inserts the new entry into the EQ, resulting in the eviction of the least recent entry (line 23).

If the evicted entry does not have a reward (indicating no request for the corresponding address within a given temporal window), CHROME assigns a reward based on the action it records, the trigger of the action, and concurrency-aware system-level feedback information (lines 24-34). If the trigger is a cache miss, bypassing is encouraged, while updating the block with the highest eviction priority (EPV_H) is encouraged if the trigger is a hit. Finally, CHROME updates the Q-value of the evicted state-action pair using the reward stored in the evicted EQ entry and the Q-value of the EQ header entry according to the SARSA algorithm [40] (lines 35-38).

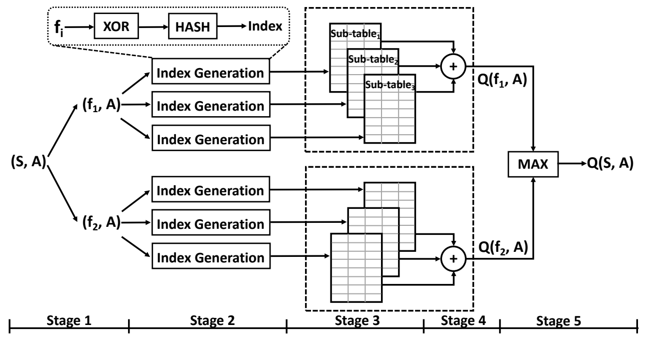

Fig. 5: Pipelined organization of Q-Table lookup.

C. Q-Table Organization

CHROME retrieves the Q-value via a table lookup, using the associated state vector and action, to exploit its learned experiences. A naive implementation uses a monolithic 2-dimensional Q-Table to record the Q-values for all possible state-action pairs. However, the storage overhead of such an implementation can be exorbitant due to the vast number of potential states. Furthermore, accessing a large monolithic table can significantly increase latency. One needs a more practical and efficient method for storing Q-values.

In CHROME, we divide the monolithic Q-Table based on the number of features in the state vector. Specifically, we partition the Q-Table into two sections, each corresponding to a feature, to record the Q-values of the feature-action pairs. To retrieve the Q-value for a given state S (which is a 2-dimensional vector of two features, f_1 and f_2, the PC signature and the page number) and an action A, CHROME queries the Q-value for each feature-action pair, i.e., $Q(f_1, A)$ and $Q(f_2, A)$, in parallel. The maximum Q-value between $Q(f_1, A)$ and $Q(f_2, A)$ is deemed the final Q-value of the state S and action A: $Q(S, A) = max(Q(f_1, A), Q(f_2, A))$. This design ensures that each CHROME action is driven by the feature offering the highest feature-action Q-value.

To reduce storage overhead and balance between resolution and generalization [6], [20], [52], we further divide each feature-action Q-Table into multiple sub-tables. Each sub-table is a two-dimensional table indexed by feature and action and stores a partial Q-value of a feature-action pair. To retrieve $Q(f_i, A)$, CHROME xors the given feature with a constant number, then uses a hash function to get the feature index for the corresponding sub-table. CHROME looks up sub-tables in parallel and obtains all the partial Q-values of a feature-action pair. The final Q-value for the feature-action pair can be computed by summing all the respective partial Q-values.

Figure 5 illustrates a five-stage pipeline for the Q-Table lookup. In the first stage, CHROME extracts the features from the given state vector and generates feature-action pairs. In the second stage, CHROME obtains the index for the corresponding sub-table of each feature-action pair. In the third stage, CHROME retrieves the partial Q-values from the sub-tables. In the fourth stage, it calculates the Q-value of each feature-action pair by summing the corresponding partial Q-values. Finally, in the fifth stage, CHROME selects the maximum value among all feature-action Q-values as the Q-value of the state-action pair.

D. EQ Organization and Q-value Update

CHROME stores a sequence of recent actions in the EQ, where each stored action is evaluated and assigned an appropriate reward based on the access patterns observed subsequently. To strike a balance between functionality and practicality, the EQ is designed to maintain an adequate number of entries to store recent actions along with their corresponding states, addresses, triggers, and received rewards, while ensuring that the total storage overhead remains acceptable. Rather than observing the cache requests from all cache sets in the LLC, CHROME randomly selects a few sampled cache sets to train the agent, a method that has been adopted earlier in other contexts [21], [26], [35], [43], [55]. Specifically, CHROME observes the cache behaviors from 64 sampled sets, and records the actions taken on these sampled sets in the EQ. To ensure the accurate evaluation of actions from these sampled sets, the EQ is organized into 64 separate first-in-first-out (FIFO) queues, each with a fixed capacity of 28 entries.

The size of the EQ is determined by balancing the need for a wider observation window of data accesses against the frequency of Q-Table updates. We provide a sensitivity analysis examining the impact of the number of FIFO entries in Section VII-F. The reward for each EQ entry is assigned either prior to or at the moment of its eviction from the corresponding queue (as discussed in Section IV).

Upon eviction of an entry from a queue, the Q-value of the evicted state-action pair is updated using the SARSA algorithm [40] based on the reward stored in the evicted EQ entry and the Q-value of the state-action pair at the head of the corresponding queue. SARSA is an on-policy reinforcement learning algorithm that estimates the Q-value based on the current policy being executed. Given the complex phase-changing workloads and the diverse access patterns and system configurations, the use of the online algorithm SARSA is considered beneficial as it allows for continuous policy updates in response to the evolving dynamics of the environment. The operation of updating the Q-value is similar to previous prediction-based schemes [21], [43] and is carried out off the critical path. We discuss the cost in more detail in Section V-G.

E. Exploration vs. Exploitation

Exploration and exploitation are important in RL. CHROME employs the ϵ-greedy method [52] to strike a balance between exploration and exploitation. CHROME randomizes its action with a small probability ϵ, exploring to gain further insights about the access pattern and the memory system. Conversely, it exploits the learned policy with a probability of 1-ϵ, selecting the action with the highest Q-value. The ϵ-greedy method ensures CHROME can sustain exploration while exploiting the existing policy to secure maximum long-term rewards.

TABLE II: Reward values and hyper-parameters.

Reward Values	R^{D}_{AC} = 20, R^{P}_{AC} = 5, R^{D}_{IN} = -20, R^{P}_{IN} = -5, $R^{\text{OB}}_{\text{AC-NR}}$ = 28, $R^{\text{NOB}}_{\text{AC-NR}}$ = 10, $R^{\text{OB}}_{\text{IN-NR}}$ = -22, $R^{\text{NOB}}_{\text{IN-NR}}$ = -10
Hyper-parameters	α = 0.0498, γ = 0.3679, ϵ = 0.001

TABLE III: Storage overhead of CHROME.

Component	Details	Overhead
Q-Table	2 features; 4 sub-tables/feature; 2048 entries/sub-table; 16 bits/entry	**32KB**
EQ	64 queues; 28 entries/queue; 58 bits/entry (state: 33 bits, action: 2 bits, reward: 6 bits, hashed address: 16 bits, trigger: 1 bit)	**12.7KB**
Metadata	EPV (2-bit/LLC block)	**48KB**
Total		**92.7KB**

F. Hyper-Parameter Tuning

Hyper-parameters, such as the learning rate (α), discount factor (γ), and exploration rate (ϵ), can significantly impact the learning efficiency of CHROME and the accuracy of its decisions. Therefore, these hyper-parameters need careful tuning. First, we define the range within which each parameter can vary. We set $\alpha \in [1\text{e}^{-9}, 1\text{e}^{0}]$, $\gamma \in [1\text{e}^{-9}, 1\text{e}^{0}]$, and $\epsilon \in [0, 1]$, providing a broad and reasonable scope for hyper-parameter tuning. Second, we divide each value range into grids, resulting in a total of 1,000 potential hyper-parameter combinations. Third, we randomly select 20 memory-intensive SPEC traces. We then evaluate the performance of CHROME across all hyper-parameter combinations. (We use a 4-core system configuration with a next-line prefetcher at L1 and a stride prefetcher at L2.) Fourth, we select the optimal hyper-parameter combination that provides the most significant geometric mean performance gain over the LRU baseline.

In Section VII-H, we show the results of a sensitivity study of the hyper-parameters. The bottom part of Table II displays the chosen values of the hyper-parameters after tuning. It is important to note that CHROME only requires a one-time hyper-parameter tuning. Once we have determined the hyper-parameters, CHROME is expected to learn online by interacting with the memory system and making decisions, without additional offline training or prior knowledge.

G. Overhead of CHROME

Table III shows the storage overhead for CHROME. The total overhead is 92.7KB, which represents only 0.75% of the capacity of a 12MB LLC in a 4-core system. This overhead is distributed across the Q-Table, the EQ, and the storage for EPVs employed in cache management. In this study, the overhead of CHROME remains constant. In particular, all cache behaviors are observed from 64 sampled sets, which does not change with the LLC capacity, effectively avoiding an escalation in storage overheads even in larger-scale systems. As shown in Table IV, CHROME has the least storage overhead compared to other SOTA cache management schemes.

When an LLC access occurs, there is a high probability that CHROME will determine the action taken by looking up the Q-Table. As a result, the Q-Table lookup operation directly impacts the decision time of CHROME. We pipeline the Q-Table lookup to reduce the decision latency (Section V-C). We use CACTI 7.0 [2] to estimate the latency for the Q-Table lookups, which comes to approximately 2 cycles in our configuration. Note that, the Q-Table operations are off the critical path, ensuring no interference with the determination of hits or misses by the cache controller. The complexity of the prediction path in CHROME is similar to that of the other SOTA prediction-based schemes [21], [35], [43]. We also use CACTI to evaluate the area and power consumption of CHROME. CHROME consumes 1.55 mm^2 of area and 76.05 mW of power in total, of which 0.30 mm^2 of area and 7.27 mW of power are used for EQ to train the RL agent – a rather modest overhead considering the significant gain in the memory performance.

TABLE IV: Storage overhead for different schemes (4-core configuration, 12-way 12MB LLC).

	Holistic	Concurrency-aware	Overhead
Hawkeye [21]	No	No	146KB
Glider [44]	No	No	254KB
Mockingjay [43]	Yes	No	170.6KB
CARE [43]	No	Yes	130.5KB
CHROME	**Yes**	**Yes**	**92.7KB**

TABLE V: Simulated system configurations.

Processor	4/8/16 cores, 4GHz, 6-wide fetch/execute/commit, 512-entry ROB, Perceptron branch predictor [24]
L1 Cache	private, 48KB D-cache, 64B line, 12-way, 5-cycle latency, 16-entry MSHR, LRU
L2 Cache	private, 1.25MB, 64B line. 20-way, 10-cycle latency, 48-entry MSHR, LRU
LLC	shared, 3MB/core, 64B line, 12-way, 40-cycle latency, 64-entry MSHR/slice
DRAM	8GB 2 channels, 2 ranks/channel, 8 banks/rank, 64-bit channel, DDR4-3200MT/s, tRP=12.5ns, tRCD=12.5ns, tCAS=12.5ns

TABLE VI: Evaluated workloads.

Suite	Workloads
SPEC 06	gcc, bwaves, mcf, milc, zeusmp, gromacs, leslie3d, soplex, hmmer, GemsFDTD, libquantum, astar, wrf, xalancbmk
SPEC 17	gcc, bwaves, mcf, cactuBSSN, lbm, omnetpp, wrf, xalancbmk, cam4, pop2, fotonik3d, roms, xz
GAP	bfs-or, bfs-tw, bfs-ur, cc-or, cc-tw, cc-ur, pr-or, pr-tw, pr-ur, sssp-or, sssp-tw, sssp-ur

VI. Evaluation Methodology

We evaluate CHROME using the cycle-accurate ChampSim simulator [16], with the version released by the 1st Instruction Prefetching Championship (IPC-1 [19]). We simulate the latest-generation Intel Alder Lake [39] multi-core processor that supports up to 16 cores. To evaluate the performance of CHROME with prefetching, we follow the methodology of 2nd cache replacement championship (CRC-2 [9]) by applying the next-line prefetcher at L1 and stride prefetcher [14], [15] at L2 as default. Table V shows the configuration parameters.

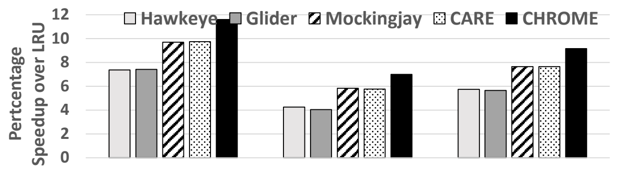

Fig. 6: Speedup for 4-core SPEC homogeneous mixes.

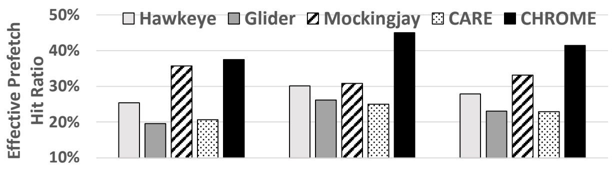

Fig. 8: Prefetch hit for 4-core SPEC homogeneous mixes.

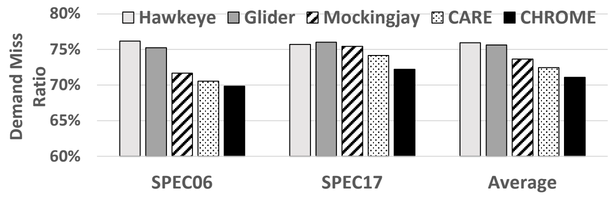

Fig. 7: Demand miss for 4-core SPEC homogeneous mixes.

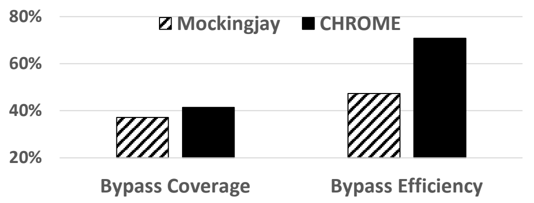

Fig. 9: Bypass coverage and bypass efficiency for 4-core SPEC homogeneous mixes.

We evaluate CHROME using a diverse set of memory-intensive workloads from the SPEC CPU2006 [46], SPEC CPU2017 [47], and GAP [3] benchmark suites. Each selected workload trace has an LLC miss per kilo instructions (MPKI) greater than 1 in the baseline system without prefetching. Table VI summarizes the workloads evaluated in this study. For the SPEC CPU2006 and SPEC CPU2017 workloads, we employ the traces provided by the 3rd Data Prefetching Championships (DPC-3 [10]), involving a total of 20 traces from 14 SPEC CPU2006 workloads and 22 traces from 13 SPEC CPU2017 workloads. For the GAP workloads, we select 5 primitive graph algorithms, including Betweenness Centrality (bc), Breadth First Search (bfs), Connected Components (cc), PageRank (pr), and Single Source Shortest Path (sssp), and 3 graph datasets, including orkut (or) [29], twitter (tw) [3], and urand (ur) [3], creating 15 distinct traces for evaluation.

For multi-core multi-programmed simulations, we use both homogeneous and heterogeneous workload mixes. For an n-core homogeneous workload setting, we test with n identical copies of a memory-intensive trace, each core executing the same trace. For an n-core heterogeneous workload setting, we randomly select n traces from all memory-intensive SPEC traces and execute a different trace on each core. We generate 150 4-core, 25 8-core, and 25 16-core heterogeneous mixes. For simulation, we warm up each core with 50M instructions from the trace, and then simulate the following 200M instructions.

To evaluate performance, LRU is selected as the baseline for comparison. We compare CHROME with four SOTA LLC management schemes: Hawkeye [21], Glider [44], Mockingjay [43], and CARE [35]. We report the results as the normalized weighted speedup over LRU, a measure commonly used for evaluating shared caches [9], [12], [43].

VII. Experiment Results

A. Performance on Four-Core Systems

Figure 6 displays the performance improvement of different cache management schemes in a 4-core system across all SPEC workloads using homogeneous workload mixes. The results show that CHROME consistently achieves outstanding performance. Specifically, with prefetching, CHROME, which employs a holistic approach to cache management, delivers an average improvement of 9.2% over the LRU baseline. In comparison, the average speedups for Hawkeye, Glider, and CARE, those without the holistic view, are 5.7%, 5.6%, and 7.6%, respectively. CHROME also adapts its caching decisions using reinforcement learning. It outperforms Mockingjay, a static method that achieves an average speedup of 7.6%.

In order to analyze the performance of CHROME in depth, we evaluate the effectiveness of CHROME in comparison with other SOTA schemes on two key metrics: LLC demand miss ratio and effective prefetch hit ratio (EPHR).

Figure 7 shows the LLC demand miss ratio comparing five cache management schemes. CHROME achieves the lowest LLC demand miss ratio at 71.1%. For comparison, the average LLC demand miss ratios for Hawkeye, Glider, Mockingjay, and CARE are 75.9%, 75.7%, 73.6%, and 72.4%, respectively. The use of online reinforcement learning in CHROME allows it to dynamically adapt to changing access patterns, ensuring that cache blocks with higher reuse potential are retained. Figure 8 displays the LLC EPHR for each scheme. We define EPHR as the ratio of prefetch hits to the total number of prefetched blocks inserted into the cache. It is a crucial metric as it evaluates how effectively prefetched blocks are utilized before eviction. CHROME leads with the highest EPHR at 41.4%. EPHR for Hawkeye, Glider, Mockingjay, and CARE comes at 27.9%, 23.0%, 33.2%, and 22.9%, respectively. This higher EPHR achieved by CHROME indicates its effectiveness in harmonizing cache replacement, bypassing, and prefetching, for improving cache performance. The ability to harmonize cache replacement, bypassing, and prefetching distinguishes CHROME from other contemporary schemes, and helps achieve superior performance, specifically its low demand miss ratio and high EPHR.

Mockingjay represents an alternative scheme that incor-

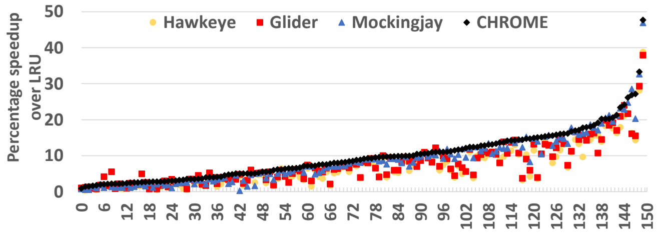

Fig. 10: Weighted speedup on a 4-core system with heterogeneous workload mixes.

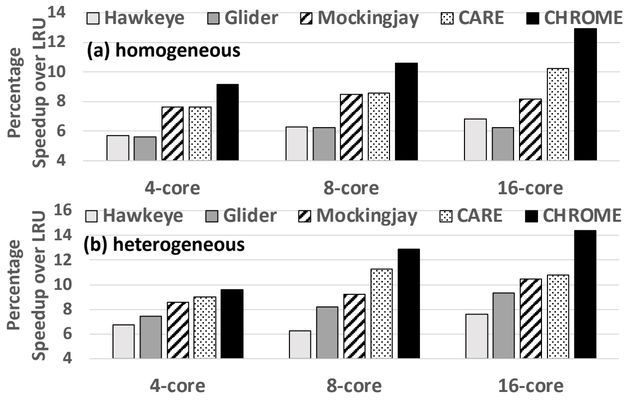

Fig. 11: Speedup for systems with 4, 8, and 16 cores (using SPEC workloads).

porates bypassing. Figure 9 shows the bypass coverage and bypass efficiency of Mockingjay and CHROME in a 4-core system. Bypass coverage quantifies the fraction of all incoming blocks that are bypassed, indicating how frequently the bypassing decisions are made. Bypass efficiency evaluates the effectiveness of the bypassing decisions by measuring the fraction of bypassed blocks that are not subsequently demanded. Our experiments show that CHROME provides higher bypass coverage and efficiency compared to Mockingjay. On average, 41.5% of incoming blocks are bypassed by CHROME, and 70.8% of the bypassed blocks are never required.

For the 4-core heterogeneous workload, we examine 150 combinations of workload mixes, as described in Section VI. Figure 10 presents the weighted speedup of Hawkeye, Glider, Mockingjay, and CHROME for all 150 cases sorted in ascending order of CHROME's performance. With prefetching enabled, CHROME provides a geometric mean speedup of 9.6% over LRU, outperforming Hawkeye, Glider, and Mockingjay with a speedup of 6.7%, 7.4%, and 8.6%, respectively. We note that CHROME demonstrates rather consistent performance improvement compared to the other SOTA schemes. For 119 out of 150 heterogeneous mixes, CHROME yields the best performance. In particular, CHROME outperforms the second-best performing scheme, Mockingjay, in 137 out of 150 cases. Evidently, CHROME can learn memory access patterns more effectively, which results in better performance gains overall.

B. Scalability

We examine the performance of CHROME as we increase the number of cores. Figure 11 summarizes the performance

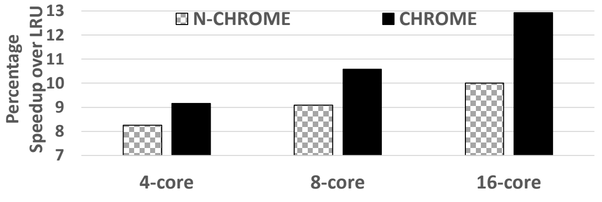

Fig. 12: Performance of CHROME and N-CHROME (using SPEC workloads).

results for both homogeneous and heterogeneous SPEC workload mixes. We make three observations from the results. First, the opportunity for improvement from cache management grows with more cores due to the increasing pressure on the LLC. Second, CHROME consistently outperforms the other SOTA schemes across all system configurations. For homogeneous workload mixes in eight (and sixteen) core systems, CHROME achieves a speedup of 10.6% (12.9%), with 6.3% (6.8%) for Hawkeye, 6.2% (6.2%) for Glider, 8.5% (8.2%) for Mockingjay, and 8.6% (10.2%) for CARE. For heterogeneous workload mixes in eight (and sixteen) core systems, CHROME achieves a speedup of 12.9% (14.4%), with 6.3% (7.6%) for Hawkeye, 8.2% (9.3%) for Glider, 9.2% (10.4%) for Mockingjay, and 11.3% (10.8%) for CARE. Last, the performance advantage of CHROME over others increases with more cores. CARE considers both data locality and concurrency when making cache replacement decisions. Consequently, it exhibits better scalability than Hawkeye, Glider, and Mockingjay. CHROME outperforms CARE by a significant margin. For homogeneous workload mixes, CHROME outperforms CARE by 1.4% on 4 cores, by 1.9% on 8 cores, and by 2.5% on 16 cores. For heterogeneous workload mixes, CHROME outperforms CARE by 0.6% on 4 cores, by 1.5% on 8 cores, and by 3.2% on 16 cores.

C. Performance without System-Level Feedback Information

To evaluate the effectiveness of the concurrency-aware system-level feedback information, we introduce a simpler version of CHROME, referred to as N-CHROME, which follows a workflow similar to that of CHROME, but lacks the awareness of the C-AMAT values for the cores. That is, N-CHROME does not differentiate situations whether the core issuing memory access is contributing to LLC obstruction. In N-CHROME, the $R_{\text{AC-NR}}$ and $R_{\text{IN-NR}}$ are set to 10 and -10, respectively, as in the non-LLC-obstruction case.

Figure 12 presents a performance comparison between CHROME and N-CHROME across all SPEC homogeneous workload mixes in systems ranging from 4-core to 16-core configurations. As expected, CHROME consistently outperforms N-CHROME for all system configurations. Moreover, the performance benefit derived from concurrency awareness increases with the number of cores. On average, CHROME provides a speedup of 9.2%, 10.6%, and 12.9% in the 4-core, 8-core, and 16-core systems, respectively, whereas N-CHROME improves performance by 8.3%, 9.1%, and 10.0%, respectively. The C-AMAT model offers accurate concurrency-

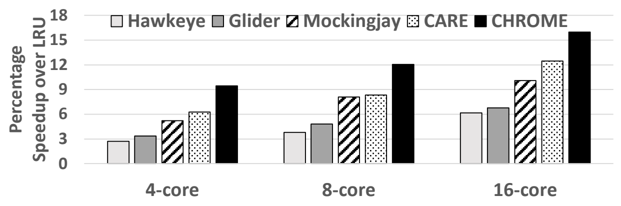

Fig. 13: Speedup for 4, 8, 16-core systems (using GAP workloads).

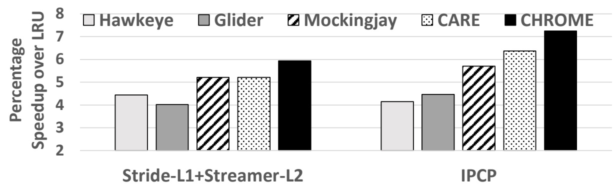

Fig. 14: Speedup with different prefetching schemes.

aware system-level feedback information, thereby enhancing the ability of CHROME to deliver robust performance. As the number of cores increases, the number of concurrent accesses in LLC also increases, thus increasing the opportunity for CHROME to learn about the environment.

D. Performance on Unseen Traces

To demonstrate the generalization of CHROME, we evaluated the performance of CHROME using GAP workloads that were not used for hyper-parameter tuning. This is to examine how CHROME reacts to unknown workloads. Figure 13 shows that CHROME consistently displays outstanding scalability and outperforms all other schemes for all configurations, even for the unseen workloads. In the 4-core system, CHROME outperforms LRU, Hawkeye, Glider, Mockingjay, and CARE by 9.5%, 6.6%, 5.9%, 4.0%, and 3.0%, respectively. In the 8-core system, CHROME achieves a speedup of 12.1% over LRU, compared with an 8.3% improvement provided by CARE, the second-best scheme among all tested schemes. In the 16-core system, CHROME improves performance by 16.0%, compared to a 12.5% improvement provided by CARE, which is again the second-best scheme in this case.

E. Performance on Different Prefetching Schemes

To evaluate the adaptability of cache management schemes, we present a performance comparison of CHROME in a 4-core system using two state-of-the-art multi-level prefetching schemes: (1) a stride prefetcher [14], [15] at L1 and a streamer prefetcher [7] at L2, a combination commonly employed in commercial Intel processors [18], and (2) IPCP [38], the winner prefetching scheme of the DPC-3 [10]. Figure 14 demonstrates that CHROME outperforms all other schemes with both prefetching configurations. When employing a stride prefetcher at L1 and a streamer prefetcher at L2, CHROME enhances performance by 5.9%, while Mockingjay, another integrated scheme, improves performance by 5.2%. When employing the IPCP prefetching scheme, CHROME attains a geometric mean speedup of 7.2% over LRU, while Mockingjay achieves a performance improvement of 5.7%. The integration of reinforcement learning greatly enhances the adaptability of CHROME across different prefetch configurations.

TABLE VII: Speedup with different FIFO sizes.

FIFO Size	12	16	20	24	**28**	32	36
Speedup(%)	6.2	7.1	7.8	8.2	**9.2**	8.0	7.5
UPKSA	911.2	884.1	857.0	830.6	**805.2**	781.4	759.1
Overhead(MB)	5.4	7.3	9.1	10.9	**12.7**	14.5	16.3

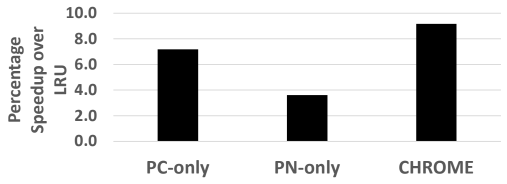

Fig. 15: Speedup with different features.

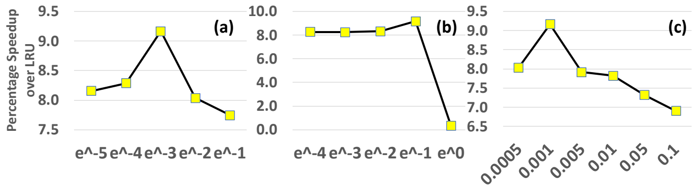

Fig. 16: Hyper-parameter sensitivity of CHROME: (a) learning rate α, (b) discount factor γ, (c) exploration rate ϵ.

F. Performance with Different EQ FIFO Sizes

Table VII shows the speedup of CHROME for all 4-core SPEC homogeneous mixes, the Q-Table updates per kilo sampled accesses (UPKSA), and the storage overhead associated with varying EQ FIFO sizes. The FIFO size strikes a balance: a larger FIFO provides a broader observation window, aiding CHROME in capturing intricate access patterns. However, this also reduces the frequency of Q-Table updates, potentially affecting the adaptability of the RL agent. Moreover, a larger FIFO brings added overhead. Optimal performance for CHROME is observed with a FIFO size of 28.

G. Performance with Different Features

Figure 15 delineates the impact of individual state features in CHROME across all 4-core SPEC homogeneous workload mixes. Utilizing the PC as the sole state feature results in a 7.2% speedup over LRU. Conversely, employing only the PN achieves a 3.6% speedup. Notably, when CHROME integrates both PC and PN, a superior speedup of 9.2% is attained. These results accentuate the combined efficacy of harnessing control-flow and data-access features. Together, they adeptly capture the nuanced memory access patterns of applications, facilitating more informed cache management decisions.

H. Performance with Different Hyper-Parameters

Figure 16 shows the overall speedup achieved by CHROME for all 4-core SPEC homogeneous workload mixes with different hyper-parameters. Figure 16(a) indicates optimal performance at a learning rate of $\alpha = 1e^{-3}$, emphasizing the balance between exploiting learned experiences and adapting to dynamic environments. Figure 16(b) reveals the best discount factor as $\gamma = 1e^{-1}$, balancing immediate and long-term rewards. Lastly, Figure 16(c) suggests that excessive exploration, beyond $\epsilon = 0.001$, can hinder performance by not sufficiently leveraging the learned policy.

VIII. Other Related Works

In our experiments, CHROME is evaluated against four state-of-the-art cache management schemes: Hawkeye [21], Glider [44], Mockingjay [43], and CARE [35]. In this section, we discuss the additional related works.

PACMan [56] mitigates prefetch-induced cache interference by altering cache insertion and hit promotion policies, distinguishing between demand and prefetch requests. SHiP++ [58] employs a history table (SHCT) to anticipate re-reference patterns of cache blocks using PCs. It refines SHiP [55] by updating the SHCT solely on the first re-reference, differentiating demand accesses from prefetches, and implementing prefetch-aware updates. PA-Hawkeye [22] evolves from Hawkeye [21] and observes that Belady's OPT algorithm falls short with prefetching. To reduce demand misses, it selectively increases prefetcher traffic. Sethumurugan et al. [42] utilize reinforcement learning offline to bolster prediction accuracy and derive insights from the learned model. While these schemes emphasize the synergy between LLC management and prefetching, they often overlook the significance of bypassing. In contrast, CHROME stands out as a holistic approach, combining cache replacement, bypassing, and prefetching. It is also an online RL-based cache management framework, using multiple program features and concurrency-aware feedback.

IX. Conclusion

This paper introduces CHROME, a novel concurrency-aware, online reinforcement learning-based holistic cache management framework. CHROME continuously learns the policy by interacting with the processor and the memory system. The nature of online reinforcement learning allows CHROME to perform well under varying system configurations and dynamic workload characteristics. CHROME makes bypassing and replacement decisions based on multiple program features and concurrency-aware system-level feedback information. Our extensive evaluations demonstrate that CHROME consistently outperforms state-of-the-art cache management schemes across different configurations, demonstrating the significant potential of CHROME for data-intensive scalable computing systems.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This research is supported in part by the National Science Foundation under Grants CCF-2008000, CNS-1956229, IIS-2331908, CNS-2310422, CNS-2152497, and by the NSF supported Chameleon testbed facility.

References

[1] A. Agarwal, K. Roy, and T. Vijaykumar, "Exploring high bandwidth pipelined cache architecture for scaled technology," in 2003 Design, Automation and Test in Europe Conference and Exhibition. IEEE, 2003, pp. 778–783.

[2] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, no. 2, pp. 1–25, 2017.

[3] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," arXiv preprint arXiv:1508.03619, 2015.

[4] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017, pp. 109–120.

[5] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems journal, vol. 5, no. 2, pp. 78–101, 1966.

[6] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 1121–1137.

[7] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," IEEE transactions on computers, vol. 44, no. 5, pp. 609–623, 1995.

[8] 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.

[9] "2nd cache replacement championship." 2017, https://crc2.ece.tamu.edu/.

[10] "3rd data prefetching championship," 2019, https://dpc3.compas.cs.stonybrook.edu/?final_programs.

[11] 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.

[12] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," IEEE micro, vol. 28, no. 3, pp. 42–53, 2008.

[13] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2017, pp. 180–193.

[14] J. W. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," ACM SIGARCH Computer Architecture News, vol. 19, no. 3, pp. 54–63, 1991.

[15] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," ACM SIGMICRO Newsletter, vol. 23, no. 1-2, pp. 102–110, 1992.

[16] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022.

[17] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," Volume 3B: System programming Guide, Part, vol. 2, no. 11, pp. 0–40, 2011.

[18] "Disclosure of h/w prefetcher control on some intel processors," https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html.

[19] "1st instruction prefetching championship," 2020, https://research.ece.ncsu.edu/ipc/.

[20] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," ACM SIGARCH Computer Architecture News, vol. 36, no. 3, pp. 39–50, 2008.

[21] 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.

[22] 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.

[23] 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.

[24] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. IEEE, 2001, pp. 197–206.

[25] 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.

[26] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2010, pp. 175–186.

[27] K. Kira and L. A. Rendell, "A practical approach to feature selection," in Machine learning proceedings 1992. Elsevier, 1992, pp. 249–256.

[28] 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.

[29] J. Kunegis, "Konect: the koblenz network collection," in Proceedings of the 22nd international conference on world wide web, 2013, pp. 1343–1350.

[30] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal bypass monitor for high performance last-level caches," in Proceedings of the 21st international conference on Parallel architectures and compilation techniques, 2012, pp. 315–324.

[31] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in International Conference on Machine Learning. PMLR, 2020, pp. 6237–6247.

[32] 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, pp. 71–89, 2021.

[33] 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.

[34] 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.

[35] X. Lu, R. Wang, and X.-H. Sun, "Care: A concurrency-aware enhanced lightweight cache management framework," in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023, pp. 1208–1220.

[36] S. Mittal, "A survey of cache bypassing techniques," Journal of Low Power Electronics and Applications, vol. 6, no. 2, p. 5, 2016.

[37] H. Najafi, J. Liu, X. Lu, and X.-H. Sun, "A generalized model for modern hierarchical memory system," in 2022 Winter Simulation Conference (WSC). IEEE, 2022, pp. 2178–2188.

[38] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 118–131.

[39] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger et al., "Intel alder lake cpu architectures," IEEE Micro, vol. 42, no. 3, pp. 13–19, 2022.

[40] G. A. Rummery and M. Niranjan, On-line Q-learning using connectionist systems. Citeseer, 1994, vol. 37.

[41] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021, pp. 291–303.

[42] 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.

[43] 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.

[44] 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.

[45] G. Singh, R. Nadig, J. Park, R. Bera, N. Hajinazar, D. Novo, J. Gómez-Luna, S. Stuijk, H. Corporaal, and O. Mutlu, "Sibyl: adaptive and extensible data placement in hybrid storage systems using online reinforcement learning," arXiv preprint arXiv:2205.07394, 2022.

[46] "Spec cpu2006 benchmark suite," 2006, http://www.spec.org/cpu2006/.

[47] "Spec cpu2017 benchmark suite," 2017, http://www.spec.org/cpu2017/.

[48] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in 2007 IEEE 13th International Symposium on High Performance Computer Architecture. IEEE, 2007, pp. 63–74.

[49] X.-H. Sun and X. Lu, "The memory-bounded speedup model and its impacts in computing," Journal of Computer Science and Technology, vol. 38, no. 1, pp. 64–79, 2023.

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

[51] X.-H. Sun, N. Zhang, B. Toonen, and B. Allcock, "Performance modeling and evaluation of a production disaggregated memory system," in The International Symposium on Memory Systems, 2020, pp. 223–232.

[52] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[53] 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.

[54] C. J. Watkins and P. Dayan, "Q-learning," Machine learning, vol. 8, no. 3, pp. 279–292, 1992.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "Rlscheduler: an automated hpc batch job scheduler using reinforcement learning," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–15.

[60] D. Zhang, D. Dai, and B. Xie, "Schedinspector: A batch job scheduling inspector using reinforcement learning," 2022.

[61] P. Zhang, R. Kannan, A. Srivastava, A. V. Nori, and V. K. Prasanna, "Resemble: reinforced ensemble framework for data prefetching," in 2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society, 2022, pp. 1168–1181.

[62] Z. Zhu, K. Johguchi, H. J. Mattausch, T. Koide, T. Hirakawa, and T. Hironaka, "A novel hierarchical multi-port cache," in ESSCIRC 2004-29th European Solid-State Circuits Conference (IEEE Cat. No. 03EX705). IEEE, 2003, pp. 405–408.