IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 10 May 2024, accepted 31 May 2024, date of publication 4 June 2024, date of current version 11 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3409228

RESEARCH ARTICLE

Reinforcement Learning-Based Cache Replacement Policies for Multicore Processors

MATHEUS A. SOUZA, (Member, IEEE), AND HENRIQUE C. FREITAS, (Member, IEEE)
Department of Computer Science, Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte 30535-901, Brazil

Corresponding author: Matheus A. Souza (matheusalcantara@pucminas.br)

This work was supported in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, in part by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under Grant 311697/2022-4, in part by Fundação de Amparo á Pesquisa do Estado de Minas Gerais (FAPEMIG), and in part by the Pontifícia Universidade Católica de Minas Gerais (PUC Minas).

ABSTRACT High-performance computing (HPC) systems need to handle ever-increasing data sizes for fast processing and quick response times. However, modern processors' caches are unable to handle massive amounts of data, leading to significant cache miss penalties that affect performance. In this context, selecting an effective cache replacement policy is crucial to improving HPC performance. Existing cache replacement policies fall short of Bélády's optimal algorithm, and we propose a new approach that leverages the coherence state and sharers' bit-vector of a cache block to make better decisions. We suggest a reinforcement learning-based strategy that learns from past eviction decisions and applies this knowledge to make better decisions in the future. Our approach uses a next-attempt method that combines the results from classic cache replacement algorithms with reinforcement learning. We evaluated our approach using the Sniper simulator and seven kernels from CAP Benchmarks. Our results show that our approach can significantly reduce the cache miss rate by 41.20% and 27.30% in L1 and L2 caches, respectively. In addition, our approach can improve the IPC by 27.33% in the best case and reduce energy consumption by 20.36% compared to an unmodified policy.

INDEX TERMS Cache replacement, coherence, multicore, reinforcement learning.

I. INTRODUCTION

Over the past decades, advancements in memory technologies for computer systems have not kept pace with processor improvements. This has widened the gap between CPU cycle time capabilities and memory access latency [1]. To address this issue, modern processors rely on memory hierarchies with caches as primary components.

The introduction of memory hierarchies has presented new challenges and research opportunities. A primary goal of this research is to explore how to use caches more efficiently. To minimize cache misses, systems must place relevant data in levels closer to the processor. Although cache misses are inevitable in memory hierarchies, strategies can be employed to optimize system performance. For instance, Bélády's optimal replacement policy for standard caches achieves the highest possible hit rates [2], [3]. However, as an offline algorithm that requires future knowledge, it cannot be implemented.

Cache replacement has been a critical research topic over the years, resulting in a variety of replacement policies and optimizations to accommodate hardware constraints, such as area and power. Well-known heuristic-based algorithms include Least Recently Used, Most Recently Used, Least Frequently Used, and random selection of ways. However, these algorithms may not be suitable for certain memory access patterns. In parallel computing scenarios that rely on caches, cache coherence becomes a concern [4]. Cache coherence transitions may force the eviction of blocks, interfering with local cache replacement policies.

In recent decades, research on cache replacement policies has evolved to incorporate modern techniques. There is a growing trend towards using prediction models to select blocks for eviction in various scenarios [5], [6], [7], [8]. These models use inputs such as frequency, reuse distance, memory

The associate editor coordinating the review of this manuscript and approving it for publication was Fabian Khateb.

access patterns, and instructions to make predictions, fitting cache replacement policies that involve decision-making processes.

This paper aims to propose new strategies for improving cache replacement policies. As future computer architectures continue to employ parallelism and shared memory hierarchies, they will still grapple with cache coherence. Thus, leveraging coherence-related behavior presents opportunities for better decision-making in cache data manipulation. For instance, consider an hypothetical scenario where two processor cores simultaneously access and modify a cache block, rendering it in a 'shared' state. Replacement policies that incorporate coherence state data can discern this contention and recognize the potential performance implications of evicting such a block. In such cases, we may prioritize the retention of 'shared' blocks to prevent performance bottlenecks stemming from frequent data transfers between cores.

Our proposed strategies focus on cache replacement policies that use the coherence state and sharing set of a cache block to make more informed eviction decisions in architectures that utilize the directory-based MESI coherence protocol at the L1 and L2 cache levels. Moreover, Reinforcement Learning (RL) has the potential to enhance cache replacement processes [9]. RL models a system with two entities–an agent and an environment–to optimize decision-making based on observed actions and environmental feedback. Our goal is to propose and evaluate algorithms that can be integrated with existing cache replacement policies to achieve better eviction decisions. Machine Learning techniques have increasingly been applied to improve computer architecture in recent years [10], [11], [12], [13].

We offer several modified versions of traditional cache replacement algorithms, employing a second-chance approach and Reinforcement Learning. We then conduct a comparative analysis of our proposals against unmodified versions of the following policies: Least Frequently Used (LFU), Least Recently Used (LRU), Most Recently Used (MRU), Not Recently Used (NRU), Pseudo-LRU (PLRU), and Static Re-Reference Interval Prediction (SRRIP).

We emphasize our main contribution: a suite of Reinforcement Learning (RL)-based eviction strategies that consider the coherence state of cache blocks and the number of cores sharing specific data. Integrating the coherence state and sharers' bit-vector into the replacement decision process is the main novelty. The RL-scheme observes past eviction decisions and leverages this information to guide future decisions while taking coherence-related behavior into account.

We outline the following contributions of our work:

- We formulate the strategy with the presence of multiple cores sharing data within the architecture.
- We adapt six well-known cache replacement policies, maintaining their core principles while incorporating our proposed modifications.
- Our approach accounts for the cache coherence mechanism and its impact on cache accesses and misses.
- Lastly, we employ Machine Learning techniques to enhance the cache replacement policies mentioned above while ensuring hardware overhead remains acceptable.

We organized the remainder of this paper as follows. Section II describes our cache replacement strategy in detail. Section III shows our experimental setup. Section IV presents the results. Section V places our work in contrast with previous ones. Finally, in Section VI, we present our conclusions.

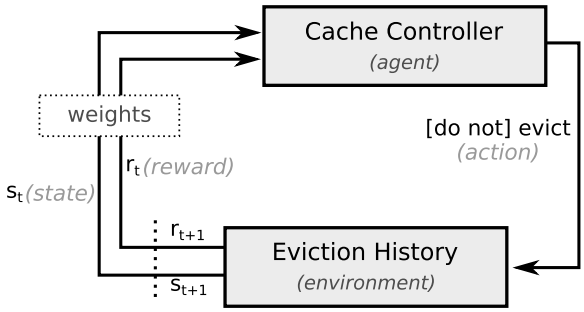

FIGURE 1. A cache reinforcement learning system.

II. COHERENCE AWARE REINFORCEMENT LEARNING-BASED CACHE REPLACEMENT

While numerous works have enhanced replacement algorithms, only a few have specifically focused on coherence [14]. Moreover, none have addressed this concern using machine learning. In this section, we discuss the novelties of our work and our contributions to the state of the art.

A. REINFORCEMENT LEARNING MODEL

Our approach utilizes Reinforcement Learning to decide whether a chosen cache block should be evicted or not. A Reinforcement Learning system consists of an agent, an environment, a policy, a reward signal, and a value function. Thus, we present our strategy formulated as a Reinforcement Learning problem. This formulation can be implemented in various ways, as RL and cache replacement share compatible theoretical principles.

Figure 1 provides an overview of our Reinforcement Learning system. The agent in our system, responsible for deciding whether to evict a cache block, is the cache controller. The environment consists of information from caches, specifically an eviction history that includes coherence and sharing information from evicted cache blocks. This environment communicates its state to the replacement agent, allowing the latter to decide which action a from the action space $A = evict, do\_not\_evict$ will be followed.

The state content comprises information from cache coherence and the sharing state of cache blocks. We formulate a vector of weights, $w_M, w_E, w_S, w_{AT}, w_{BT} \in \mathbb{R}$, which forms our value function $V^{\pi}(s)$. Here, M, E, and S represent

Modified, Exclusive, and Shared, corresponding to different coherence states. Additionally, AT and BT refer to 'above the threshold' and 'below the threshold,' respectively.

The threshold is a binary value indicating whether a block is shared by more than a specific number of cores. In our experiment, we set the threshold to half of the total number of cores. For instance, in a 16-core processor, if a block's bit vector is set to 9 or more cores, it is considered above the threshold; otherwise, it is considered below. We further explain how we update and utilize these values. While the weights are real numbers ($\mathbb{R}$), future improvements may consider the use of natural numbers ($\mathbb{N}$) for potentially better performance.

The environment reacts to the chosen action with a reward signal $R \in \mathbb{R}$ and an updated state $s \in S$, which will only be processed in the future. In fact, our reward design follows a binary approach: a negative reward (-1) is assigned when a cache miss occurs for a block that was previously evicted and recorded in the eviction history, indicating a poor eviction decision; and a positive reward ($+1$) is granted when a cache miss occurs for a block not in the eviction history, suggesting the past eviction decisions were effective. Our policy $\pi(s)$ postpones the use of states by at least one unit of time t. The main drawback of this decision-making process relies on the fact that we can only determine whether a block should be evicted or not when we need it again in the future.

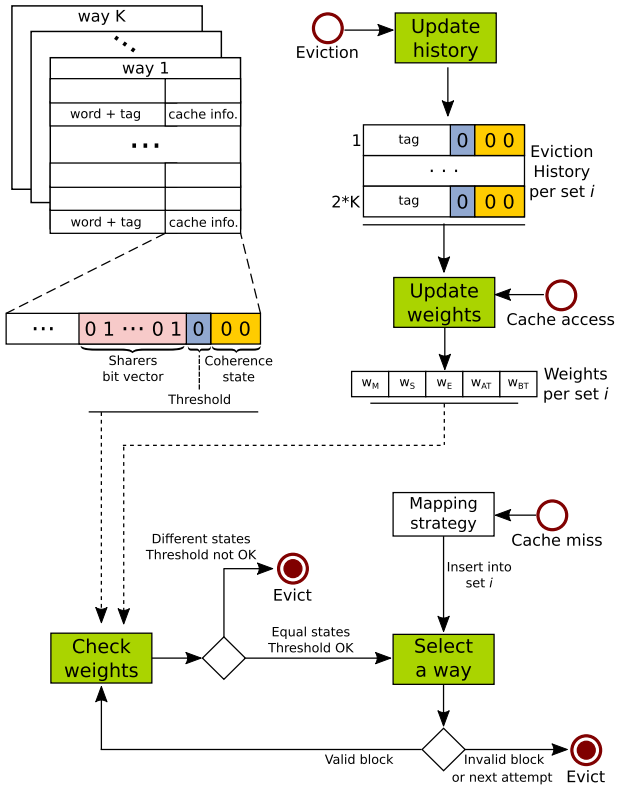

FIGURE 2. The reinforcement learning-based eviction strategy.

B. THE EVICTION HISTORY TABLE AND COHERENCE WEIGHTS

The vector of weights of our approach is updated concerning an eviction history that is part of the policy in the Reinforcement Learning system. Figure 2 depicts the entities involved in this process and provides an overview of the strategy's operation. There are three starting points which triggers important steps of our approach, such as the history and weight's update phases.

When an eviction occurs (the 1st starting point), information from the evicted block is added to the eviction history of each set. If the new cache block has a corresponding tag in the history, that entry is removed. The eviction history table keeps track of the eviction decisions made by the cache replacement policy. Each entry contains the cache block's tag; a bit indicating if it was above (0) or below (1) the threshold; and two bits representing the coherence state. The table is used to update the weight vector during the learning process, allowing the policy to make better eviction decisions in the future. To update the weights of our learning-based cache replacement model, we store the memory tag to ensure the uniqueness of the entries in the history. If the history is full, we remove the least recently updated entry.

Whenever a cache access occurs (the 2nd starting point), we update the weights based on whether the requested block is in the table or not. If the accessed cache block is not found in the eviction history table, this indicates a favorable eviction decision thus, the reward is positive. Conversely, if the block is found, indicating a suboptimal eviction previously, the reward is negative. The reward is determined by the presence or absence of the cache block in the eviction history table, influencing future weight adjustments that guide the eviction policy. The first distribution represents the coherence state, with the highest probability of being a good eviction decision (w_M, w_E, w_S). In this case, each weight is a ratio of the number of entries in this state in the table to the total number of entries. Thus, the sum of these weights always equals 1. The second distribution indicates which direction to observe (above or below) concerning the given threshold for the number of sharers (w_{AT}, w_{BT}). It is also a ratio of the number of entries above or below the threshold.

In our work, during the initial stages of computation when no probability distribution exists, we set the Exclusive coherence state as the most likely candidate for eviction. This approach results in reduced coherence traffic. We applied the same principle of minimizing traffic when initializing the algorithm by evicting blocks with a sharers bit-vector count below the threshold. Consequently, the exploration-exploitation trade-off is naturally resolved during computation, as this information changes independently of the reward.

A larger table size increases accuracy but also increases hardware space overhead and vice-versa. We chose a relatively small table size – twice the associativity – to reduce hardware space overhead while still providing enough information to make accurate eviction decisions. The table is partitioned into sets, with each set corresponding to a set in the cache. This allows for local information to be used in the

eviction decision, as the coherence state and sharer count are likely to be similar within a cache set. Then, to execute the strategy, we need $2 \cdot K \cdot N$ entries in the entire cache, where K is the associativity and N is the number of sets the cache contains.

C. THE REPLACEMENT STRATEGY

The replacement process of a cache block begins with a cache miss (the 3rd starting point), followed by the cache mapping, as depicted in Figure 2. Then, a candidate way and its block are chosen, and we decide whether to evict by comparing the vector of weights to the cache block. If the block information corresponds to high-probability elements in the vector, the block is evicted; otherwise, the block is given a second chance. For instance, the LRU selected a block in the Modified coherence state. However, the w_M value is the highest in the vector. Thus, this block is kept in the cache, and the second LRU block is chosen.

D. A NEXT-ATTEMPT APPROACH

Our primary goal is not to devise an entirely new cache replacement algorithm. Instead, we assert that the sharing state or the number of sharers of a cache block can be used to enhance existing algorithms. Our proposal is to evaluate whether the choice made by already known algorithms meets specific criteria. To achieve this, we employ an approach called the next-attempt. This approach is similar to the one employed in the Second-Chance Page Replacement algorithm. However, we do not use a reference bit to determine if the block or page should be given a second chance.

The idea begins with the traditional algorithm running unaltered to select a cache block. We then examine the chosen block using our heuristics. If it satisfies a specific condition, the strategy evicts it. Otherwise, the algorithm retains this block in the cache and selects the second option provided by the traditional algorithm.

For instance, LRU and PLRU algorithms select the least recently used block within the cache set. If this block does not meet a given condition from our strategy, we retain it in the cache. Consequently, the algorithm selects the second least recently used cache block. Similarly, the next-attempt chooses the second most recently used block when using MRU.

For LFU, the next-attempt selects the cache block with the second lowest frequency of use. The cost of this strategy in these algorithms relies on tracking the first and second victims at each iteration. The original algorithm already maintains this information since it must select the next candidate when an eviction occurs.

NRU and SRRIP use the Re-Reference Interval Prediction; thus, they do not track a sequence of accesses or frequencies that can be properly sorted. However, remember that they employ a bit-wise strategy to select the victim. Therefore, we address this issue by altering the NRU-Bit or the RRPV to a value that ensures the algorithm runs another iteration without selecting this cache block. In these two algorithms, the added cost is the need to rerun an iteration of the algorithm in the worst-case scenario.

III. EXPERIMENTAL METHODOLOGY

In this section we present the experimental methodology employed to evaluate our strategies. We focus on the selection of tools, workloads, and evaluation criteria used to assess the performance and effectiveness of the proposed strategies.

A. SIMULATOR AND TOOLS

To effectively assess our cache replacement strategies, we need an environment that offers a flexible computer architecture capable of modeling a multicore system with multiple cache levels and support for cache coherence protocols. Moreover, the environment should enable the execution of various applications and collect a wide range of metrics to analyze the behavior of the strategies in diverse parallel workloads.

Considering these requirements, we have chosen the Sniper simulator [15] for conducting our experiments. Sniper is an execution-driven multicore simulator that is built upon the interval core model. This model allows for individual clock and timing management while maintaining a window of instructions for each simulated core, providing a realistic representation of a multicore system.

The simulator leverages the Pin dynamic instrumentation framework [16] to collect execution traces from multi-threaded applications, effectively filtering out irrelevant traces and focusing on regions of interest. This capability allows for more accurate analysis of the cache replacement strategies.

Additionally, Sniper incorporates the McPAT framework [17] into its core, which we utilize for measuring power and energy consumption. This integration offers valuable insights into the energy efficiency of the cache replacement strategies under investigation, contributing to a more comprehensive evaluation of their overall performance.

B. SIMULATED ARCHITECTURE DESIGN

We evaluate our proposed strategies using a simulated 16-core processor based on the x86 micro-architecture, with each core operating at a frequency of 2.66 GHz. The cache hierarchy consists of three levels. The Last Level Cache (LLC) is the L3 cache, which is a shared 16MB, 16-way set-associative cache. As we do not evaluate the L3 cache in our experiments, we configure the LLC to satisfy all data requests at this level. The second level comprises private 8-way set-associative L2 caches, each with a capacity of 256kB. Finally, the first level consists of private L1 caches, with separate 32kB caches for data and instructions. Both the L1 and L2 caches are 8-way set-associative. We set the cache block size to 64 bits, maintaining consistency across the cache hierarchy. We did not use a prefetcher in our evaluation to isolate the effects of our proposed approach on cache replacement policy and to provide a fair comparison with the baseline policies.

TABLE 1. Comparison of cache replacement policies.

Policy	Bits required	On cache access
LFU	$N \cdot K \cdot log_2(K)$	Update frequency
LRU	$N \cdot K \cdot log_2(K)$	Update LRU order
MRU	$N \cdot log_2(K)$	Update MRU-bits
NRU	$N \cdot K$	Update NRU-bits
PLRU	$N \cdot (K-1)$	Update tree bits
SRRIP	$N \cdot 2 \cdot K$	Update RRPV

The simulated architecture employs the Modified, Exclusive, Shared, Invalid (MESI) directory-based cache coherence protocol. Since the protocol uses only four states, we require two bits per cache entry to represent them in the directory. Although one could use the MOESI or MESIF protocols, they would necessitate three bits per cache entry due to the additional states. In this work, we exclusively investigate the MESI protocol.

We consider the following well-known cache replacement policies for evaluation: LFU, LRU, MRU, NRU, PLRU, and SRRIP. To assess the complexity of each algorithm, we examine the number of bits required in the cache and the operations executed during cache accesses. Table 1 presents a modified version of the comparison made by Al-Zoubi et al. [18], with the addition of MRU, LFU, NRU, and SRRIP algorithms.

We adapt the comparison to reflect our implementations, detailing storage costs and actions executed during cache accesses for each algorithm. The number of bits required for each policy is calculated as a function of the number of sets (N) and the cache associativity (K).

We apply each of our cache replacement strategies simultaneously to both L1 and L2 caches. For example, if L1 caches utilize the SRRIP strategy, L2 caches will also employ SRRIP. This approach is implemented to mitigate the local replacement hazard [19].

C. WORKLOADS

A primary objective of our work is to assess the performance of our proposed strategies when executing parallel applications on the modified architecture. These applications must generate workloads that produce cache accesses and misses within the cache hierarchy. Additionally, they should involve parallel workloads with inter-thread communication, which necessitates memory usage and generates coherence operations.

We use the seven kernels from the CAP Bench suite [20] for our experiments. CAP Bench is a benchmark suite designed to serve as a baseline for designing, programming, evaluating, and learning about low-power manycore architectures. The benchmark suite also offers a version for x86 architectures, implemented in OpenMP, and has been slightly adapted for use with the Sniper simulator – we delimited the region of interest, speeding up simulations..

The CAP Bench kernels encompass various application design aspects, such as parallel patterns, with a focus on manycore architectures even in their OpenMP version. Consequently, the benchmark and the workloads it generates are well-suited for our simulation methodology.

TABLE 2. Kernels available in CAP bench.

Kernel	Parallel pattern	Job Type	Memory access intensity	Task load
FAST	Stencil	CPU and Memory	High	Irregular
FN	MapReduce	CPU	Low	Regular
GF	Stencil	CPU	Average	Regular
IS	Divide and Conquer	Memory	High	Irregular
KM	Map	CPU and Memory	High	Irregular
LU	Workpool	Memory	High	Regular
TSP	Workpool	CPU	Low	Irregular

Table 2 provides an overview of the characteristics of all CAP Bench kernels. The parallel pattern column indicates the parallel programming strategy employed by each kernel. The job type categorizes the kernel based on the resource predominantly required during its execution. Memory access intensity reflects the frequency of memory access events. Finally, the task load is considered regular if there is a balanced workload distribution among threads.

- FAST is an image corner detection method implemented following the Stencil parallel pattern. The workload input we use is an image of 8192×8192 pixels.
- FN computes the *abundancy* of numbers in an interval, which is a mathematical property. The interval used in our experiments is $8 \times 10^6 + 1$ to $8 \times 10^6 + 2^{14}$.
- GF is a filter algorithm that also deals with images using matrix convolution. The input image has 8192×8192 pixels, and the algorithm uses masks of 11×11 pixels.
- IS is an algorithm to sort a certain amount of integer numbers using the bucket sort strategy. In our experiments, we set an amount of 2^{25} integer numbers as input size.
- KM is a well-known clustering algorithm. The input data is $2^{14}\mathbb{R}^{16}$ points and 512 centroids.
- LU is a matrix decomposition algorithm, which follows the Workpool parallel pattern. The workload in our experiments consists of a 1536×1536 matrix.
- Finally, the TSP solves the Traveling-Salesman Problem in a traveling scenario with 17 towns.

D. PERFORMANCE METRICS

We use various metrics to evaluate the behavior and effectiveness of our cache replacement approaches. We decided to use IPC and cache miss rates due to their common usage in related works, although these works also use speedup and MPKI. The primary metric we consider is the cache miss rate from data caches, as the replacement algorithms directly impact

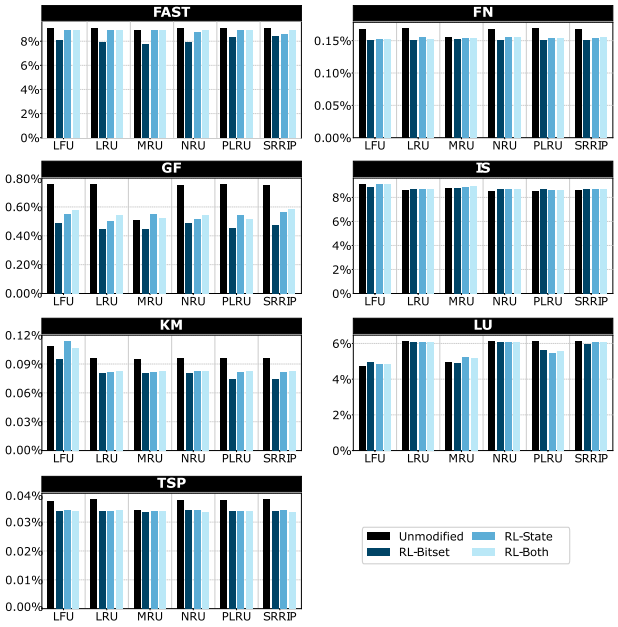

FIGURE 3. L1 cache miss rate.

the cache's block presence. For our experiments, we do not consider the misses from instruction caches.

Additionally, we measure the coherence traffic by comparing the number of invalidates and write-backs between different replacement strategies. The coherence protocol can impact performance. Thus, reducing the drawbacks of coherence while favoring its benefits is crucial. Therefore, we assess our strategies' performance based on coherence traffic.

Another metric we consider is the instructions per cycle (IPC). IPC is a general metric and corresponds to the instruction throughput in the processor. Since memory access instructions take longer to execute than processor instructions, and cache levels farther from the processor are even slower, the IPC will vary significantly depending on the cache miss rates. Thus, IPC is an important metric for measuring the performance of our strategies.

Lastly, we measure energy consumption, which is the product of time to solution and power consumed during kernel execution. An improvement in cache usage leads to less computing time and reduced use of cache elements that consume more power. Therefore, we consider energy consumption as a crucial metric in our experiments.

IV. RESULTS

In this section, we present the results obtained after simulating our Reinforcement Learning strategies.

To evaluate these strategies, we followed a two-step process. In the first step, we used the coherence state and the sharer's count as information for the Reinforcement Learning algorithm. We refer to these strategies as RL-State and RL-Bitset, respectively. In the second step, we evaluated the use of both types of information simultaneously (RL-Both).

To establish a baseline, we compared the performance of our proposed policies against unmodified policies. Therefore, except for cache miss, our charts present results in terms of gains or losses (increase or decrease) against the respective unmodified algorithm. We compare each policy using our three strategies, and we present one kernel per chart.

We evaluated our policies using several metrics. First, we analyzed the cache miss rate. Next, we measured the coherence write-backs and invalidates caused during computation. Finally, we present the instructions per cycle and energy results.

A. CACHE MISS RATE

Figure 3 presents the overall comparison between strategies concerning the decrease in L1 cache miss rate. We observe that our Reinforcement Learning-based strategies outperform unmodified policies in most cases. With FAST, the RL-Bitset yields up to a 12.91% decrease in miss rate when using the LRU policy.

For FN, GF, and TSP, our approaches exhibit fewer improvements with the MRU policy, occasionally presenting higher miss rates. The most significant L1 miss rate improvement for FN is 11.69% using the PLRU/RL-Bitset strategy. For GF, the best result is a 41.2% improvement with the RL-Bitset approach and LRU policy. Lastly, an 11.73% reduction in miss rate is observed for TSP using the LRU/RL-Bitset strategy.

Despite minor differences, our strategies also reduce miss rates for KM, with the exception of the RL-State approach with LFU. The best L1 miss rate improvement for this kernel is 22.88% using the SRRIP policy and RL-Bitset technique.

For the LU kernel, the PLRU policy stands out, as our strategies reduce miss rates more significantly than with other policies. For example, the PLRU/RL-State approach achieves a 10.18% reduction in miss rate for this kernel.

IS exhibits different behavior compared to other kernels. For this kernel, only the RL-Bitset strategy yields any improvement – the miss rate with LFU and RL-Bitset is reduced by 2.07%. Our Reinforcement Learning-based strategies fail to improve the other policies. We hypothesize that our strategy might be prematurely evicting shared cache blocks that are unmodified but expected to be modified in the near future. This is supported by an increase in invalidation coherence traffic, as illustrated in Figure 5 from Section IV-B. Such inefficiencies in predicting the lifespan of cache blocks within the IS kernel exacerbate performance issues, leading to suboptimal cache management.

Figure 4 provides an overview of the decrease in L2 cache miss rates. Miss rate variations for FAST and GF at this cache level are minimal. Nevertheless, the most substantial miss rate reduction for FAST is 0.37% using the RL-Bitset strategy and LFU policy. For GF, we observe negative results for LRU with the same strategy, which is 0.68% worse than the baseline.

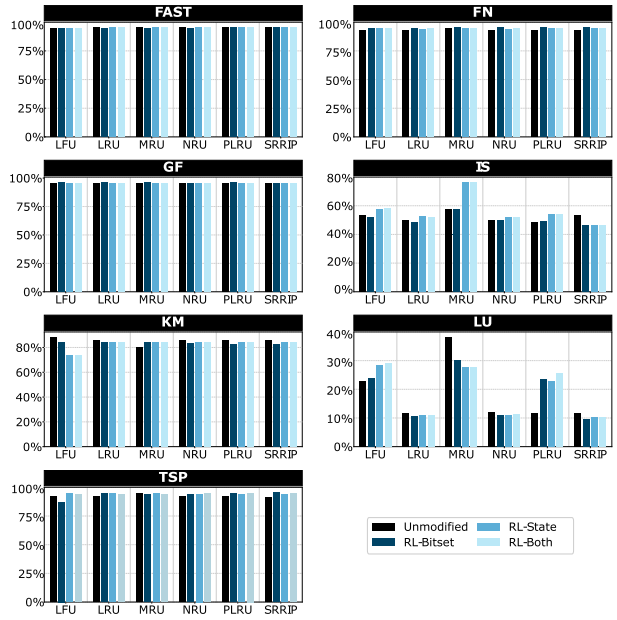

FIGURE 4. L2 cache miss rate.

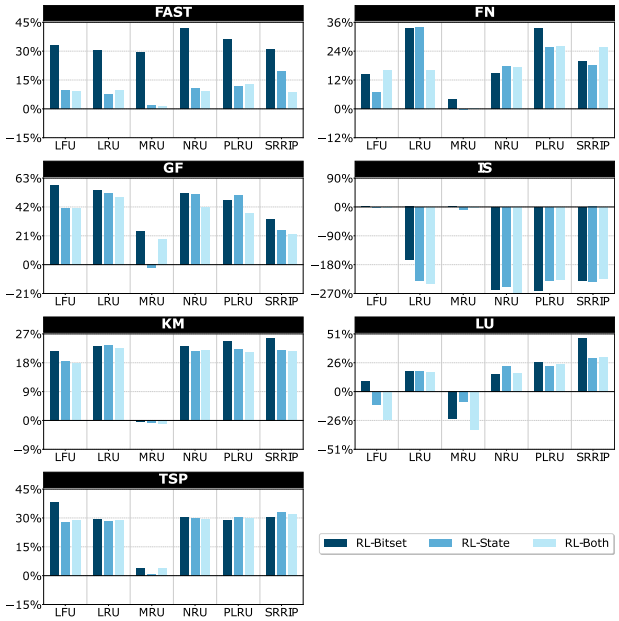

FIGURE 5. Coherence invalidates decrease.

Referring to Figure 4, FN, GF, and TSP show almost no miss rate reductions in L2 cache, regardless of the applied strategies. This pattern is consistent with previous observations, where improvements in L1 cache miss rates lead to fewer L2 cache accesses, thus increasing the rate at this level. Nonetheless, the RL-State approach yields the best L2 miss rate reductions for the SRRIP policy when running GF (0.15%). Additionally, with TSP, the RL-Bitset improves the LFU policy by reducing the L2 miss rate by 5.24

For IS, we highlight the LFU, LRU, and SRRIP policies, in which our RL-Bitset approach attains better results than the baseline. Specifically, for SRRIP, all Reinforcement Learning-based approaches outperform the original SRRIP. The RL-Both strategy is the best, reducing the miss rate by 13.63%.

With the KM kernel, the most significant variations occur with LFU. The RL-State and RL-Both strategies achieve around 16.2% miss rate reductions over the baseline with this policy. However, no improvements are observed for this kernel using the MRU policy.

Conversely, our techniques reduce the L2 miss rates with MRU and SRRIP for the LU kernel. The SRRIP/RL-Bitset approach reduces the miss rate by 17.09% compared to the unmodified policy. The MRU/RL-State approach achieved 27.30% of miss rate reduction.

B. COHERENCE OPERATIONS

We collected the invalidates from the RL-based approaches and depicted these values in Figure 5.

FAST, FN, GF, KM, and TSP present substantial improvements when our Reinforcement Learning-based techniques are employed. This indicates that our strategies more effectively determine which blocks to keep in the cache, consequently reducing the need to invalidate blocks that may be present in other caches. However, such improvements are not observed with the MRU policy.

More specifically, RL-Bitset performs best for FAST, regardless of the policy it is employed in. In this kernel, this approach improves the NRU policy, resulting in 41.85% fewer invalidates. For FN, the most significant improvement is a decrease of 34.01% with the LRU/RL-State. GF is the kernel with the highest reduction, achieving up to 57.77% fewer invalidates than the baseline with the LRU policy using the RL-Bitset strategy.

The RL-Bitset strategy is also the best for KM and TSP. We observe 27.16% fewer invalidates with SRRIP and 37.96% with LFU for these kernels, respectively. We also see some improvements in LU, with up to 47.03% fewer invalidates when using the SRRIP policy with the RL-Bitset strategy.

Figure 6 displays the improvements concerning coherence write-backs. Most of the kernels exhibit favorable results, although they are relatively smaller than those of invalidates. The best results for each kernel, detailed below, are observed with the RL-Bitset approach. Firstly, for FAST with the LRU policy, we achieve 12.57% fewer write-backs than the unmodified policy. The best case for FN is 6.52% with PLRU, as well as for GF, which demonstrates 20.98% fewer operations than the unmodified PLRU. IS exhibits the most significant decrease among all kernels, at 35.56% when using the LRU with the RL-Bitset approach.

Continuing with the RL-Bitset approach, the coherence write-backs are reduced by 27.16% using the SRRIP policy for the KM kernel. For LU, the LFU policy shows the most significant decrease, at 17.87% over the baseline.

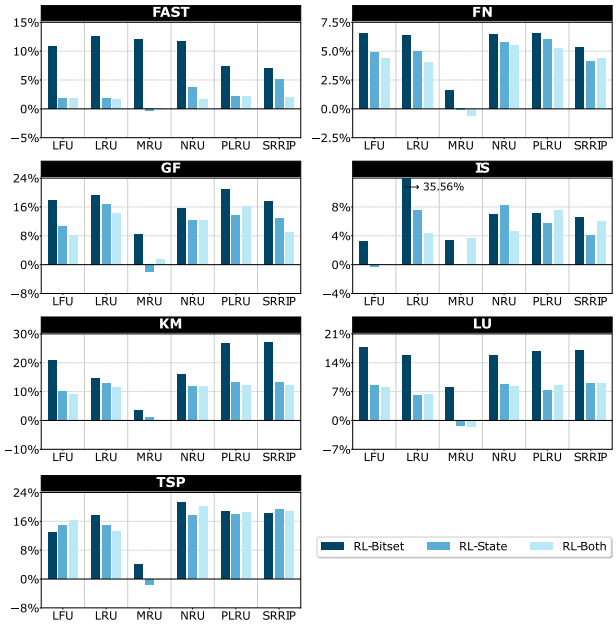

FIGURE 6. Coherence write-backs decrease.

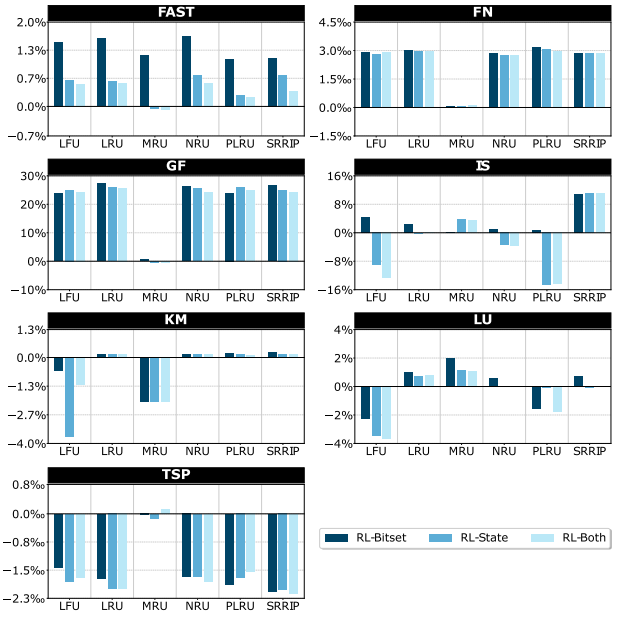

FIGURE 7. Instructions per Cycle (IPC) increase.

Lastly, the TSP kernel, when run in the architecture with the NRU policy, displays 21.24% fewer write-backs than the baseline.

C. INSTRUCTIONS PER CYCLE

To evaluate the performance of our Reinforcement Learning-based approaches, we present their observed IPC in Figure 7. FN and TSP show less significant variations in IPC than other kernels when with our strategies. Nonetheless, the IPC for these kernels follows the cycles spent behavior. Since TSP is compute-bound, the main computational workload focuses more on processing rather than frequent memory accesses. The slight decrease in IPC could indeed be attributed to the minor overhead incurred from the RL strategy's 'check weights' phase. In compute-bound scenarios, the relative impact of cache management on performance is often less pronounced because the CPU spends more time calculating than fetching data from memory.

FAST and GF kernels demonstrate notable improvements over the baseline with all Reinforcement Learning-based strategies, except for the MRU policy. The best result for FAST is observed with the NRU policy using the RL-Bitset approach, which has a 1.68% higher IPC than the baseline. For GF, the LRU policy is where our RL-Bitset strategy can achieve the most significant IPC improvement, reaching up to a 27.33% increase. These IPC improvements also follow the decrease in the number of cycles. Likewise, the reduced cycles spent are due to the improvements in miss rates and coherence operations for these kernels. Moreover, FAST and GF deal with the same input type and follow the same parallel pattern, though they are different. GF performs fewer memory accesses and takes longer to execute than FAST [20]. Consequently, the impact of our strategies on GF is more significant, resulting in higher IPC gains.

Concerning the IS kernel, the best results are obtained with SRRIP, owing to the cycles spent with each strategy. This policy shows up to 11.22% higher IPC than the baseline when the RL-Both strategy is employed. However, we also observe some IPC decreases in this kernel using the PLRU policy, which is around 14.8%. PLRU dynamically approximates LRU behavior without tracking the exact order of cache lines. Misguided evictions due to inaccurate RL predictions about coherence states can disrupt PLRU's tree balance. A similar issue applies to NRU, which relies on binary markers to identify blocks that have not been recently used. This also applies to NRU when trying to mark the blocks that have not been recently used, and our strategy ignores this binary information. Such evictions may remove blocks that are not the least recently used but still hold future value. Therefore, employing the RL-Bitset approach in this scenario could yield better results.

Similar to TSP, KM exhibits more pronounced decreases than increases. For example, the IPC when using LFU is reduced by 3.7% with the RL-State technique compared to the baseline. Focusing on the eviction of MRU or LFU blocks is suboptimal in scenarios where data points accessed recently still hold significant value for upcoming computations, such as recalculating centroids. There is also a rise in invalidation traffic, indicating frequent invalidation of cache lines still needed by other cores or threads. This could be due to the dynamic relocation of centroids in KM, which affects the coherence protocol and results in higher synchronization overhead. Thus, our strategy exacerbates existing problems in this MRU and LFU scenario for KM,

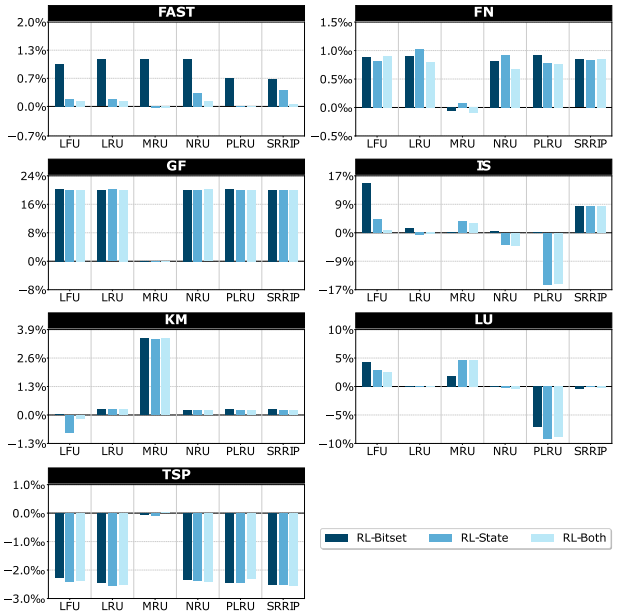

FIGURE 8. Energy consumption decrease.

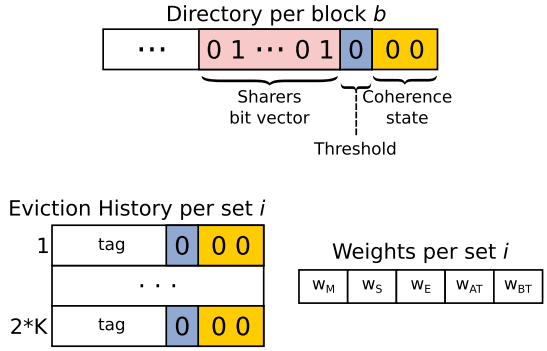

FIGURE 9. Bits added to the cache architecture.

contributing to minor processing overhead that adversely impacts IPC.

In contrast to other kernels, with LU, the MRU policy demonstrates the best improvements in IPC. We observe a 1.9% higher IPC with this policy using the RL-Bitset approach. However, as in KM, we cannot improve the LFU policy using our Reinforcement Learning-based strategies. Furthermore, we notice the opposite behavior when using the MRU policy.

D. ENERGY CONSUMPTION

Figure 8 presents the decrease in energy consumption that our strategies can achieve. Consistently, FAST, FN, and TSP do not exhibit significant changes in energy consumption, as their IPC and cache miss results also do not vary much when we modify the cache replacement strategy. We have already mentioned that higher IPC values correspond to lower energy consumption. For these kernels, as well as for GF and IS, the energy results exhibit a behavior similar to the IPC.

FAST demonstrates up to 1.11% less energy consumed than the baseline, using the RL-Bitset approach. For FN, the best result is with RL-State applied to LRU, which is 0.1% lower than the unmodified policy. GF shows almost equal results for all policies and strategies, which is around 20.1%. The best result for IS is observed in the LFU/RL-Bitset configuration, with 15.09% less energy consumed. We also highlight the SRRIP policy for IS, which can be improved by all of our Reinforcement Learning-based strategies. Finally, TSP displays almost no improvements in energy consumption, being worse by a margin of approximately 0.23% than the baseline.

Regarding KM and LU, we find that the IPC decrease is not sufficient to negatively impact energy consumption. Generally speaking, the improvements for KM are minimal (3.48% using MRU and the RL-Bitset approach). For LU, the best result is 4.35% less energy consumed with the LFU/RL-State strategy.

E. HARDWARE OVERHEAD

To discuss our strategies' hardware overhead, let us recall that we used the directory-based protocol. Figure 9 displays the entities with additional bits to enable our strategy. First, consider the per-block directory entry. Each one contains a 16-bit vector to track sharers, one bit to identify if the block has surpassed the threshold of sharers, and two bits for the coherence state. As a result, each directory entry has at least 19 bits in our 16-core processor experiment.

Now, consider the eviction history table. There is one table for each cache set, and each table contains a number of entries equal to twice the associativity denoted by K. The number of sets is denoted by N, which varies according to the cache size. T is the number of bits required for the *tag*, present in each cache block. Consequently, all the eviction histories occupy a total of $2KN$ entries in the cache. Each history entry contains the tag T, which we use to make them unique. Additionally, they have the past evictions threshold and state information. Considering these bits, we have at most $2KN(T+3)$ bits to implement the eviction histories in the cache.

Lastly, the vector of weights comprises five real numbers ($\mathbb{R}$). To implement this in real hardware, we should use natural numbers ($\mathbb{N}$) for better performance and occupation. A simpler strategy would be to use a single bit that indicates if blocks in that state or threshold should be kept in the cache.

We emphasize that we can reduce this space overhead according to the cache configuration. For instance, if we increase the block size, the cache will have fewer sets, reducing the number of eviction history tables. As a last resort, we could also explore a global strategy, i.e., a single eviction history and vector of weights for all sets. This approach would significantly reduce the space overhead.

TABLE 3. Comparison between related work on ML-based cache replacement.

#	ML technique	Approach	Input data	Architecture	Workload	Mean miss decrease vs. LRU*	Mean speedup vs. LRU*
[21]	MDP	online	hit/miss	multicore	benchmark complete run	–	10.8%
[22]	Classification	offline	program counter	multicore	benchmark complete run	17.00%	8.4%
[23]	Perceptron	offline	several features	single core	benchmark sampling	22.20%	8.3%
[7]	Imitation learning	offline	memory accesses	single core	benchmark sampling	19.62%	–
[6]	LSTM and SVM	offline/online	program counters	multicore	benchmark sampling	8.90%	8.1%
[5]	Regret minimization	online	reuse distance frequency	cache-only	synthetic traces	25.11%	–
[8]	Reinforcement learning	offline	memory accesses	multicore	benchmark sampling	–	3.5%
[24]	Deep reinforcement learning	offline	other policies info.	not informed	benchmark sampling	–	–
Our work	Reinforcement learning	online	cache coherence info.	multicore	benchmark complete run	6.57%	4.7%

* The values in "Mean miss decrease vs. LRU" and "Mean speedup vs. LRU" are averaged, each under different architectures and workloads, e.g., synthetic traces, as reported in related studies. Due to variations in the experimental conditions, direct comparisons of these averages should be made cautiously. Each study's context and setup significantly influence the reported outcomes.

In terms of processing overhead, we assess the core operations of our strategy, depicted in Figure 2. The 'Update history' operation transfers $T + 3$ bits – tag and state information of the cache block – to the eviction history upon every eviction. The 'Update weights' operation adjusts the values in the weight vector to rebalance the decision-making environment. Our method is tailored to asynchronous hardware architectures, which mitigates potential processing delays, ensuring these updates occur seamlessly without impacting the way selection.

Furthermore, the 'Check weights' phase incurs minimal processing overhead. This phase involves a rapid comparison of stored weights against the current cache block's coherence state and sharing threshold to determine eviction suitability. These factors collectively ensure that our RL scheme enhances cache performance without significant computational cost, making it suitable for high-performance computing environments.

V. RELATED WORK

This section presents related work in cache replacement that somehow applies Machine Learning-based strategies. For example, the Economic Value Added (EVA) is the strategy proposed by Beckmann et al. [21] The authors claimed that cache replacement could be modeled as a Markov Decision Process due to how it works. They adopted the idea and proposed using the EVA of cache blocks, which is the difference between the number of hits a cache block should yield and its average hit count and occupancy.

Hawkeye is the policy proposed by Jain and Lin [22]. The authors used Bélády's algorithm to learn the optimal solution from past accesses. The strategy uses a trainer that receives cache access history and computes the optimal algorithm. Next, the predictor uses this information and the PC to classify cache blocks into cache-friendly or cache-averse. The policy consults the predictor in every cache insertion and promotion.

The Multiperspective Placement, Promotion, and Bypass (MPPPB) [23] technique use a multiperspective reuse predictor. The MPPPB predictor captures several characteristics from the application in execution, producing many perspectives. Furthermore, the predictor also uses information from the LRU stack to perform the training steps, besides the program counter (PC) information. Thus, with those inputs, the predictor produces highly accurate results for eviction decisions.

Machine Learning-based cache replacement can also use an Imitation Learning approach [7]. The main idea is to mimic some behavior from examples. This work consisted of a model to learn cache access patterns using Bélády's algorithm.

Another work also proposed an offline tool using Reinforcement Learning to learn access patterns [8]. It evaluated the learned model and implemented the cache replacement strategy focusing on specific features. The main goal was to check the reuse distance, predicted based on the distance of the cache blocks used in the past. When the predicted reuse distance was higher than the reuse distance of a cache block, the policy kept this block in the cache.

Similarly, we can mention the Long Short-Term Memory (LSTM) approach [6]. This work trained an LSTM model to extract patterns from the program counters (PC) of load and store instructions. The model is trained offline using Bélády's algorithm and then used to build an online SVM-based hardware predictor to perform the cache replacement.

LeCaR is a general framework that uses the regret minimization technique [5]. It uses two data structures for each cache block to track their frequency (LFU) and recency (LRU) of use. Then, it chooses which strategy to follow based on what it has learned from the data structures.

Last but not least, the *Catcher* framework was built to learn the relationship between different replacement policies [24]. This work evaluated the probability distribution of several replacement algorithms and workload distributions using Deep Reinforcement Learning. The framework trains an end-to-end cache replacement policy based on requested addresses concerning recency (LRU) and frequency (LFU). It learns from LRU and LFU results and applies these approaches.

Table 3 shows characteristics of past work on these Machine Learning-based cache replacement policies, posi-

tioning ours against them in the state of the art. Our work is complementary to the previous work. First, we observe a predominant use of recency and frequency information on them, which is naturally used in ours, as shown in Section II. Next, none of them are aware of cache coherence in the replacement policy, which is relevant information to the eviction decision. In addition, most of them are different in terms of Machine Learning techniques. One work used Reinforcement Learning with an offline approach [8], which might be a paradox considering the learning purposes. Thus, there are gaps to explore in this state of the art, which we do in our paper.

In terms of results, the works have reported improvements in IPC of 14.7% [6], 4.86% [8], and 9.1% [23]. Also, an 8.9% reduction in cache miss [6] and 1% energy overhead [22]. These results are shown here to position ours against related work. Table 3 presents the "Mean miss decrease vs. LRU" and "Mean speedup vs. LRU." These metrics, reflecting averages from various applications used in related work, provide broader context while acknowledging the methodological differences. However, the direct comparison against related work is challenging due to significant methodological differences - the number of workloads and their types, architectures, and experimental setups. It is important to note that most of these studies did not consider multicore architectures or coherence traffic, which are central to our approach. Additionally, the predominance of offline learning methods in previous works contrasts with our online learning approach, which integrates coherence information - a crucial aspect they lack. Nevertheless, the potential for integrating our strategy with others in future research could enhance the effectiveness of cache replacement policies. Our results affirm the benefits of incorporating coherence information and employing a next-attempt strategy, underscoring the robustness and applicability of our method.

VI. CONCLUSION

Caches are critical components for the performance of modern processors, and the replacement policy they implement is essential. With the coherence problem affecting most modern multicore processors, finding a replacement policy that considers information about the sharing state of a cache block is crucial. In this paper, we presented a Reinforcement Learning approach that uses the sharing state and the number of cores that share a cache block to improve the cache replacement.

We implemented our approach on three well-known replacement policies, namely LFU, LRU, MRU, NRU, PLRU, and SRRIP, and observed significant improvements in cache miss rates, instructions per cycle, and energy consumption. Our RL-Bitset approach, which considers the number of cores that share a cache block, achieved up to 41.20% reduction in L1 cache miss rate and up to 17.09% reduction in L2 cache miss rate, leading to up to 27.33% IPC gains and 20.10% energy consumption reduction over the baseline. Overall, our work demonstrates that a reinforcement learning-based approach that considers the coherence state and sharers' bit-vector of a cache block can significantly improve HPC performance.

While our approach used information from coherence and the sharer's bitset locally within cache sets, future research could explore using this information globally. Additionally, we suggest exploring the use of our strategies in other memory environments, such as virtual memory page replacement. Then, it would be interesting to investigate how our approach performs on applications such as web servers and big data, which run codes repeatedly on different data sets, and whether knowledge learned could be sustained between each run.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, doi: 10.1145/216585.216588.

[2] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966, doi: 10.1147/sj.52.0078.

[3] A. Jain and C. Lin, *Cache Replacement Policies* (Synthesis Lectures on Computer Architecture), vol. 14. Cham, Switzerland: Springer, 2019, doi: 10.1007/978-3-031-01762-9.

[4] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 2nd ed. Cham, Switzerland: Springer, 2020, doi: 10.1007/978-3-031-01764-3.

[5] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving cache replacement with ML-based LeCaR," in *Proc. 10th USENIX Conf. Hot Topics Storage File Syst.* USA: USENIX Association, 2018, p. 3.

[6] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 413–425, doi: 10.1145/3352460.3358319.

[7] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proc. 37th Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 6237–6247.

[8] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 291–303, doi: 10.1109/HPCA51647.2021.00033.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, U.K.: A Bradford Book, 2018.

[10] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1919–1928.

[11] A. Zouzias, K. Kalaitzidis, and B. Grot, "Branch prediction as a reinforcement learning problem: Why, how and case studies," in *Proc. 2nd Workshop Mach. Learn. Comput. Archit. Syst.*, 2021, pp. 1–6.

[12] D. D. Penney and L. Chen, "A survey of machine learning applied to computer architecture design," in *Proc. Int. Workshop AI-Assisted Design Archit.*, 2019, pp. 1–14.

[13] N. Wu and Y. Xie, "A survey of machine learning for computer architecture and systems," *ACM Comput. Surv.*, vol. 55, no. 3, pp. 1–39, Mar. 2023, doi: 10.1145/3494523.

[14] B. Panda and S. Balachandran, "CSHARP: Coherence and SHaring aware cache replacement policies for parallel applications," in *Proc. IEEE 24th Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2012, pp. 252–259, doi: 10.1109/SBAC-PAD.2012.27.

[15] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 1–25, Aug. 2014, doi: 10.1145/2629677.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, Jun. 2005, doi: 10.1145/1064978.1065034.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[18] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite," in *Proc. 42nd Annu. Southeast Regional Conf.* New York, NY, USA: Association for Computing Machinery, Apr. 2004, pp. 267–272, doi: 10.1145/986537.986601.

[19] M. Zahran, "Cache replacement policy revisited," in *Proc. Workshop Duplicating, Deconstructing, Debunking*, 2007, pp. 1–8.

[20] M. A. Souza, P. H. Penna, M. M. Queiroz, A. D. Pereira, L. F. W. Góes, H. C. Freitas, M. Castro, P. O. A. Navaux, and J. Méhaut, "CAP bench: A benchmark suite for performance and energy evaluation of low-power many-core processors," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 4, Feb. 2017, Art. no. e3892, doi: 10.1002/cpe.3892.

[21] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 109–120, doi: 10.1109/HPCA.2017.43.

[22] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 78–89, doi: 10.1109/ISCA.2016.17.

[23] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 436–448.

[24] Y. Zhou, F. Wang, Z. Shi, and D. Feng, "An end-to-end automatic cache replacement policy using deep reinforcement learning," in *Proc. Int. Conf. Automated Planning Scheduling*, vol. 32, 2022, pp. 537–545, doi: 10.1609/icaps.v32i1.19840.

MATHEUS A. SOUZA (Member, IEEE) received the tech degree in systems analysis and development from FABRAI, Belo Horizonte, Brazil, in 2007, and the M.Sc. and Ph.D. degrees in informatics from the Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Belo Horizonte, in 2015 and 2021, respectively. He is a member of the Computer Architecture and Parallel Processing Team (CArT). Since 2022, he has been a Professor with PUC Minas. He was a Visiting Doctoral Student with the Laboratoire TIMA, Grenoble, France, from September 2018 to February 2019. His research interests include high-performance computing, parallel programming, shared and distributed memory hierarchies, the application of machine learning to computer architecture, operating systems, and compilers. He is a member of Brazilian Computer Society (SBC).

HENRIQUE C. FREITAS (Member, IEEE) received the B.S. degree in computer science and the M.Sc. degree in electrical engineering from the Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Belo Horizonte, Brazil, in 2000 and 2003, respectively, and the Ph.D. degree in computer science from the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, 2009. Since 2004, he has been a Professor with PUC Minas. From 2015 to 2016, he was a Visiting Researcher with INRIA, Université Grenoble Alpes, Grenoble, France, funded by the Brazilian Research Council (CNPq). His research interests include computer architecture, high-performance computing, parallel computing, and heterogeneous computing. He is a member of Brazilian Computer Society (SBC).