2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00033

Designing a Cost-Effective Cache Replacement Policy using Machine Learning

Subhash Sethumurugan
Electrical and Computer Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
Email: sethu018@umn.edu

Jieming Yin
Electrical and Computer Engineering
Lehigh University
Bethlehem, PA 18015
Email: yin@lehigh.edu

John Sartori
Electrical and Computer Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
Email: jsartori@umn.edu

***Abstract*—Extensive research has been carried out to improve cache replacement policies, yet designing an efficient cache replacement policy that incurs low hardware overhead remains a challenging and time-consuming task. Given the surging interest in applying machine learning (ML) to challenging computer architecture design problems, we use ML as an offline tool to design a cost-effective cache replacement policy. We demonstrate that ML is capable of guiding and expediting the generation of a cache replacement policy that is competitive with state-of-the-art hand-crafted policies.**

In this work, we use Reinforcement Learning (RL) to learn a cache replacement policy. After analyzing the learned model, we are able to focus on a few critical features that might impact system performance. Using the insights provided by RL, we successfully derive a new cache replacement policy – Reinforcement Learned Replacement (RLR). Compared to the state-of-the-art policies, RLR has low hardware overhead, and it can be implemented without needing to modify the processor's control and data path to propagate information such as program counter. On average, RLR improves single-core and four-core system performance by 3.25% and 4.86% over LRU, with an overhead of 16.75KB for 2MB last-level cache (LLC) and 67KB for 8MB LLC.

I. Introduction

Caches are an important component in modern processors. The effectiveness of a cache is largely influenced by its replacement policy. An efficient cache replacement policy can effectively reduce off-chip bandwidth utilization and improve overall system performance. There is a large body of prior work on cache replacement policies; however, designing cost-effective cache replacement policies is still challenging for chip designers, especially under stringent hardware constraints.

Cost-effectiveness is becoming increasingly important, as Moore's Law has slowed down and Dennard scaling has ended. A cost-effective cache replacement policy should be able to reduce misses-per-kilo-instructions (MPKI) without introducing significant hardware modification and storage overhead. Commonly used Least Recently Used (LRU) and Re-Reference Interval Prediction (RRIP) polices [12] incur minor hardware overhead for storing the recency bits or re-reference counters. However, these static heuristic-driven policies are only effective for a limited class of cache access patterns. Using program counter (PC) information, state-of-the-art replacement policies are capable of capturing dynamic phase changes in cache access patterns, and they can effectively reduce the MPKI for a wide spectrum of workloads [11], [14], [24], [30], [34]. Figure 1 compares the last-level cache hit rate among different replacement policies, in which Belady is the theoretical optimal. Unsurprisingly, the PC-based policies (SHiP, SHiP++, and Hawkeye) outperform non-PC-based policies (LRU and DRRIP) in almost all benchmarks.

TABLE I
Hardware overhead for different replacement policies in a 16-way 2 MB cache

Policy	Uses PC	Overhead
LRU	No	16KB
DRRIP [12]	No	8KB
KPC [19]	No	8.57KB
MPPPB [14]	Yes	28KB
SHiP [30]	Yes	14KB
SHiP++ [34]	Yes	20KB
Hawkeye [11]	Yes	28KB
Glider [24]	Yes	61.6KB
RLR (this work)	**No**	**16.75KB**

Unfortunately, incorporating PC into the replacement policy not only requires additional storage overhead but also involves significant modifications to the processor's control and data path. Accessing PC at the LLC requires propagating PC through all levels of the cache hierarchy, including widening the data path, modifying cache architecture to store PC, adding extra storage for PC in the Issue Queue, Reorder Buffer (ROB) and Load/Store Queue (LSQ), and more [19]. More costly than the overhead of implementing PC-based policies is the fact that these changes require an overhaul of the entire processor pipeline, which would require significant design and verification overheads. Given the extensive modifications required, chip manufacturers have thus far been unwilling to implement PC-based replacement policies, favoring instead incremental design enhancements with similar performance/overhead tradeoffs that do not require significant redesign and verification. For example, for the same memory overhead of implementing SHiP++ in LLC, a designer could implement DRRIP in LLC and also increase L1 size by 12KB. Given that increasing L1 size may provide similar or even better benefits without requiring significant redesign and verification, processor manufacturers have as yet not implemented PC-based replacement policies. Table I summarizes the hardware overhead of different cache replacement policies.

How can we improve the cache replacement policy without using PC? We realized this is a difficult challenge. To answer

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00033

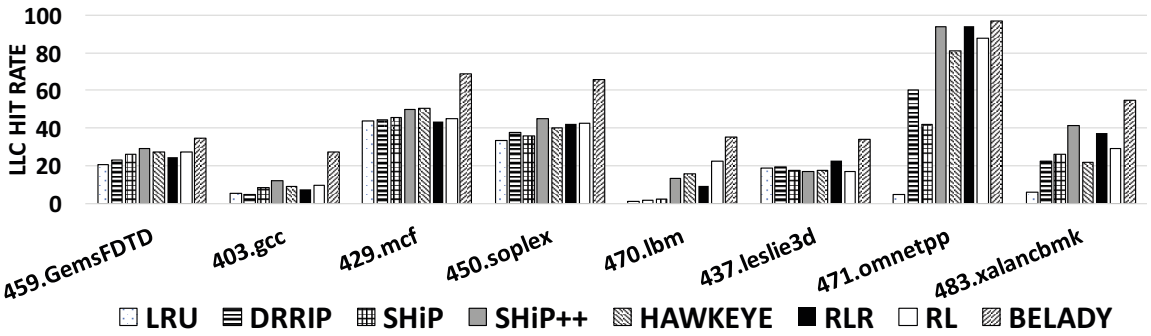

Fig. 1. LLC hit rate comparison (Belady is the theoretical optimal).

this question in a cost-effective way in terms of reducing product development time, we turned to machine learning for help. Machine learning is a useful tool to augment human intelligence and expedite the chip design process, and computer architects have been using ML to advance computer architecture designs. There has already been work on utilizing ML to improve branch predictors [13], memory controllers [22], reuse prediction [28], prefetchers [35], dynamic voltage and frequency scaling management for network-on-chip (NoC) [9], [36], and NoC arbitration policy [32], [33].

In this work, we explore the capability of ML in designing a cost-effective cache replacement policy. Specifically, we use reinforcement learning (RL) to learn a last-level cache (LLC) replacement policy. The RL algorithm takes into consideration a collection of features that can be easily obtained at the LLC without modifying the processor's control and data path. After successfully learning a replacement policy that achieves good performance, we analyze the learned policy by applying domain knowledge, with the goals of distilling useful information, verifying the important features, understanding the relative importance of each feature, and gaining insights into how these features interact. Guided by ML, we frame a cost-effective cache replacement policy – Reinforcement Learned Replacement (RLR). Overall, RLR does not require heavy modification to the CPU microarchitecture and outperforms LRU and DRRIP for most evaluated benchmarks. Figure 1 compares LLC hit rate for several replacement policies, including a policy learned by RL and our static adaptation (RLR) based on RL.[1] The performance of the RL policy is better than LRU, DRRIP, SHiP, and Hawkeye in most benchmarks, while it is marginally lower than SHiP++. RL performance can be improved by including PC-based features in the feature set, but one goal of our work is to design a cost-effective replacement policy that does not rely on PC at the LLC. The performance of RLR shows the effectiveness of the insights learned from RL.

II. Related Work

There has been considerable work on designing efficient cache replacement algorithms [7], [16], [17], [20], [23], [31]. In this section, we discuss some of the related prior work.

[1]Due to the computational complexity of running RL simulations and the need to look at future accesses, RL and BELADY simulations are run in a python-based simulator; the rest of the policies use ChampSim. Further details are explained in Section III.

Glider: Glider [24] uses a Support Vector Machine (SVM)-based hardware predictor for cache replacement. Initially, an offline attention-based long short-term memory (LSTM) model is used to improve prediction accuracy. Then, the authors interpret the offline model to gain insights and hand craft a feature that represents a program's control-flow history. Then, a simple linear learning model is used to match the LSTM's prediction accuracy. The hardware implementation of the policy requires a Program Counter History Register and an Integer SVM table.

Hawkeye: Hawkeye [11] uses a PC-based predictor to determine whether a cache line is cache-friendly or cache-averse. On a miss, the policy first chooses to evict cache-averse lines. If no cache-averse cache lines are found in the set, the oldest accessed cache line is chosen for eviction.

SHiP: Signature Based Hit Prediction (SHiP) [30] replacement policy predicts re-reference characteristics of cache lines from a PC-based signature. SHiP has a Signature History Counter Table (SHCT) that maintains a counter (SHCTR) for each PC signature. Cache lines inserted by PCs with non-zero SHCTR are assigned a Re-Reference Prediction Value (RRPV) of 2, while other cache lines are inserted with a RRPV of 3.

SHiP++: The SHiP++ replacement policy [34] enhances SHiP by inserting cache lines accessed by PCs with maximum SHCTR value with a RRPV of 0, training the SHCT table only on the first re-reference of a cache line, inserting cache lines from writeback accesses with a RRPV of 3, assigning a separate PC signature for prefetch accesses, and making prefetch-aware RRPV updates on a cache line re-reference.

Counter based replacement policy: Kharbutli *et al.* [18] propose a counter-based approach, where each cache line is equipped with counters to track events such as the number of accesses to the set between two consecutive cache line accesses or the number of cache line accesses. When the counter reaches a threshold, the line is eligible for replacement. The policy also uses a PC-based prediction table to retain counters for evicted cache lines.

All of the above policies correlate the reuse behavior of cache lines to the PCs. However, accessing PC at the LLC adds significant hardware overhead to the architecture. The width of the data path must be increased to propagate PC through all levels of the cache hierarchy. In addition, the miss status holding registers (MSHR) also need to track the PC information. These modifications exacerbate energy

and communication overheads. Furthermore, pipeline design must be modified to propagate PC through all stages, as well as adding extra storage at Load/Store Queue before accessing the memory system. While many of the prior works lack the justification of the hardware and processor design/verification overhead associated with incorporating PC in cache replacement, it is questionable whether the benefits will outweigh the overhead and complexity. As a result, we avoid using PC in our replacement policy.

KPC: Kill the Program Counter [19] proposes an integrated data prefetching and replacement policy that avoids using the PC. While our work aims to design an effective replacement policy given an existing prefetcher, KPC designs a custom prefetcher (i.e., KPC-P) to complement their replacement policy. KPC-P uses prediction confidence to estimate reuse distance for prefetched cache lines and determine the cache level in which to insert them. The KPC cache replacement policy (i.e., KPC-R) uses two global counters to adapt to the dynamic program phase and decide whether to insert a cache line in LRU (RRPV=3) or near LRU (RRPV=2) positions in the replacement stack. KPC avoids L2 cache pollution by not inserting prefetched lines with low prediction confidence in L2; however, all prefetched lines are inserted in LLC. As described in Section III-B2, we avoid cache pollution from prefetched cache lines in LLC by evicting non-reused prefetched cache lines sooner than cache lines from other access types.

The following replacement policies use past accesses of a cache line to predict its future access behavior. Each policy predicts a cache line's reuse characteristic using a certain metric. On a miss, all cache lines in the accessed set are compared using the metric, and the cache line with the farthest reuse characteristic is chosen for eviction.

PDP: Protecting Distance based Policy (PDP) [6] protects all lines in LLC until the number of accesses to the set (after the line insertion or access) reaches a threshold, known as Protecting Distance (PD). On a miss, an unprotected cache line is evicted. If no unprotected cache lines are found, either the cache line with the minimum number of set accesses is evicted or the access is bypassed. A dedicated special-purpose processor executes a search algorithm periodically to compute the optimal threshold. Hit rates are estimated for threshold values less than 256, and the threshold value with the best hit rate is chosen. In our work, we derive a much simpler method to predict reuse distance (like PD in [6]) based on insights gained from the ML model.

EVA: The Economic Value Added (EVA) metric [4] characterizes the difference between expected and average hits. A cache line's EVA depends on its age, and the cache line with the lowest EVA is evicted. However, the policy does not account for non-demand accesses, such as prefetch accesses. These additional accesses may skew the correlation between a cache line's age and its EVA.

RWP: Read-Write Partitioning (RWP) [16] is a replacement policy that dynamically partitions the cache into clean and dirty partitions to reduce the number of read misses. On a miss, a victim is selected from one of the partitions, based on predicted partition size and the actual partition size in the corresponding set.

Inter-reference Gap Distribution Replacement [27] uses time difference between successive references of a cache line to attach a weight to it. On a miss, the cache line with the smallest weight is evicted. Das *et al.* [5] propose using a cache line's age (since last access) to estimate its hit probability under an optimal replacement policy. On a miss, the line with the lowest hit probability is evicted. Keramidas *et al.* [15] combine the usage of reuse distance and PC. The policy uses a sampler to compute reuse distances for selected cache lines. The computed reuse distances and the PCs of load/store instructions that accessed the selected cache lines are used to predict reuse distances for other cache lines. On a miss, the cache line with largest predicted reuse distance is evicted.

III. Machine Learning-Aided Architecture Exploration

Reinforcement Learning is a machine learning paradigm in which an agent tries to navigate through an environment by choosing an action from a set of allowed actions [25]. Using the suggested action, the environment moves from the current state to the next state and meanwhile generates a reward as a feedback to the agent. The agent trains itself to maximize cumulative reward. In this process, the agent learns a policy that selects the optimal action in a given state. One way to keep track of the optimal action for a given state is to maintain a table for all state-action pairs. However, it could be infeasible to implement such a table when the state and action space is large. In such a scenario, a neural network can be used as a function approximator in lieu of a table.

RL has the potential to learn a theoretical optimal policy, given that the effects of actions are Markovian [29]. Because RL has the ability to adapt to dynamic changes in the environment and handle the non-trivial consequences of chosen actions, it is a good fit for the cache replacement problem. We pose cache replacement as a Markov Decision Process (MDP), where an agent makes replacement decisions. Given a cache state, the replacement decision made by the agent moves the cache to a new state. The agent is assigned a reward based on how close the replacement decision is to BELADY (optimal). In our framework, we train a neural network using RL algorithms to learn a replacement policy. Although the learned policy can be efficient, we do *not* want to build a neural network in hardware, due to power, area, and timing constraints. Instead, we analyze the neural network and use the insights gained from the neural network to derive a replacement algorithm that is feasible to implement in hardware. In this section, we describe our simulation framework and architecture exploration flow in detail.

A. RL-based Simulation Framework

At high level, our simulation framework consists of two parts: trace generation and RL training. We use ChampSim [3] from the 2^{nd} Cache Replacement Championship (CRC2) to generate LLC access traces. The trace file comprises a record of $\langle$*PC, Access Type, Address*$\rangle$ for each LLC access. Access types include load (LD), request for ownership (RFO),

TABLE II
LIST OF FEATURES CONSIDERED BY THE RL AGENT

Classification	Feature	Description
Access Information	offset	Lower order 6 bits of accessed address
	preuse	Set accesses since last access to the accessed address
	access type	Type of access (LD, RFO, PR, WB)
Set Information	set number	Set that was accessed
	set accesses	Total number of set accesses
	set accesses since miss	Set accesses since last miss to the set
Cache Line Information	offset	Lower order 6 bits of cache line address
	dirty	Dirty bit of the cache line
	preuse	Set accesses between last two accesses to the cache line
	age since insertion	Set accesses since cache line insertion
	age since last access	Set accesses since last access to the cache line
	last access type	Type of last access to the cache line (LD, RFO, PR, WB)
	LD access count	Number of load accesses to the cache line
	RFO access count	Number of read-for-ownership accesses to the cache line
	PF access count	Number of prefetch accesses to the cache line
	WB access count	Number of write-back accesses to the cache line
	hits since insertion	Number of hits to cache line since its insertion
	recency	Order of cache line access with respect to other cache lines in the set

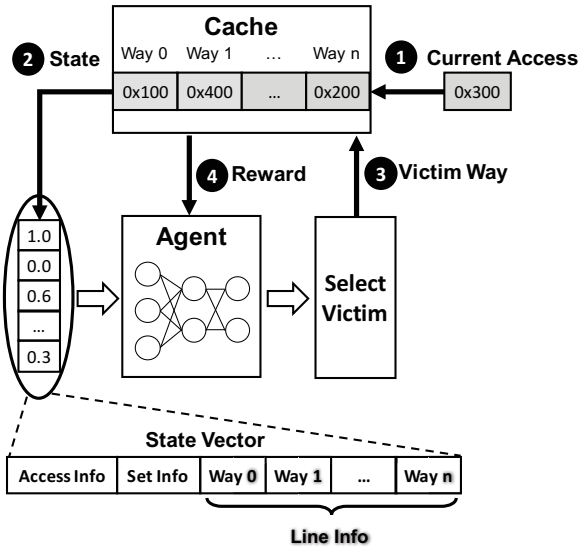

Fig. 2. Simulation framework overview.

prefetch (PR), and writeback (WB). We use LRU as the default replacement policy to ensure that the generated traces are not biased towards other policies that we compare against.

The trace is fed into a Python-based cache simulator that includes an RL agent to make replacement decisions. The cache simulator uses the same LLC configuration as ChampSim and populates the LLC based on the accessed addresses. Each cache line is associated with a collection of cache states, as described later in this section. The simulation framework is shown in Figure 2. On a hit, the cache simulator updates the cache states and moves on to the next access. On a non-compulsory miss, the cache simulator interacts with the agent to make a replacement decision ❶. Information regarding the missed access and the accessed set is sent to the agent in the form of a state vector ❷. The agent evaluates the state vector and generates an output vector of n elements for an n-way set associative cache ❸. Each element in the output vector corresponds to a way in the cache set, and the value represents how beneficial it is (from the agent's perspective) if a certain way is chosen for eviction. The cache simulator then makes a replacement decision based on the output vector generated by the agent; meanwhile, a numerical reward is generated and sent to the agent for further training ❹. Below we explain the critical components is detail.

State Vector: LLC state vector contains information required to make a replacement decision. We segregated LLC state into three classes of features: a) access information that describes the current access to the cache; b) set information that describes the set that is being accessed; and c) cache line information that describes each cache line in the set that is being accessed. On every LLC access, statistics of the accessed set are updated. For example, the counter *set access* is incremented on every access to the set. As another example, the counter *set access since miss* is incremented on every hit to the set and reset to zero on a miss. Similar counters are maintained for every cache line in the set. The cache line counters are reset in the event of an eviction to start counting for the newly inserted cache line. Cache lines are also augmented to store other information such as their recency, last access type, dirty bit and other relevant features. The entire feature list representing LLC state is listed in Table II. Categorical features such as *last access type* are one-hot encoded. Numerical features such as *access count* are normalized by their respective maximum values and represented as a fractional value between 0 and 1. The only exception is the feature *offset*, for which we use a 6-bit binary representation (assuming 64-byte cache lines). For a 16-way set associative LLC, we represent a state vector using 334 floating point values.

Agent: The agent consumes the state vector and generates an output vector of size equal to the set associativity of the LLC. Each value in the output vector indicates the estimated quality of choosing the cache line in the corresponding way as a victim. In this work, we use a neural network to estimate the quality. After extensive exploration on neural network

architecture and hyperparameter tuning, we chose to use a multi-layer perceptron (MLP) with one hidden layer, because it is simple enough for interpretation but performs almost as well as denser networks. We also found that *tanh* activation for the hidden layer and *linear* activation for the output layer yielded better performance than other combinations. The neural network has 334 input neurons, 175 hidden neurons and 16 output neurons (because of the 16-way LLC). On every cache miss, the simulator queries the agent to select a victim. In our simulation framework, there is only one neural network for victim selection for all sets of the LLC. This is similar to following a common replacement policy for all sets. Designers can choose to use multiple agents by training them using different combination of cache sets.

Replacement Decision: The agent returns a vector of n values, one for each cache way (e.g., 16-element vector for a 16-way cache). The replacement decision is made by an ε greedy approach [21], in which we choose the victim with the maximum value with a probability of $1-\varepsilon$ and randomly select a victim with a probability of ε. Random actions explore new trajectories and expand the search space. In our experiment, we found that an ε value of 0.1 yielded better performance than other ε values.

Reward: The reward steers the agent towards learning a more optimal replacement policy, so reward function must be chosen carefully. A theoretically optimal replacement policy, such as Belady, replaces the cache line that has the farthest reuse distance among lines in a set. To allow the agent to learn this behavior, a positive reward is returned when the agent makes a good decision and evicts the cache line with the farthest reuse distance. A negative reward is returned when the agent evicts a cache line with a lesser reuse distance than the cache line that is inserted in cache, since the evicted cache line would hit sooner than the inserted cache line if retained in cache. A neutral reward is awarded when any other cache line is evicted. Only the optimal replacement decision is assigned a positive reward, differentiating it from the other decisions and allowing the agent to learn a near-optimal policy faster.

Training: For training, we use a technique called experience replay [21]. Each replacement decision is stored as a transaction in a replay memory. A transaction is represented by a tuple of $\langle$*state, action, next state, reward*$\rangle$. A replay memory is a circular buffer with a limited number of entries, and the oldest transaction is overwritten by a new transaction. Instead of using the most recent transaction, the neural network is trained using a batch of randomly sampled transactions from replay memory. Experience replay breaks the similarity of subsequent training samples, which in turn reduces the likelihood of the neural network from being directed into local minima. In addition, experience replay allows the models to learn the past experience multiple times, leading to faster model convergence and reduced training time.

B. Insights from Neural Network

Neural networks have the ability to achieve better performance by learning a favorable policy after sufficient training. However, it is likely not cost effective to implement a neural network in LLC due to significant power and area overheads.

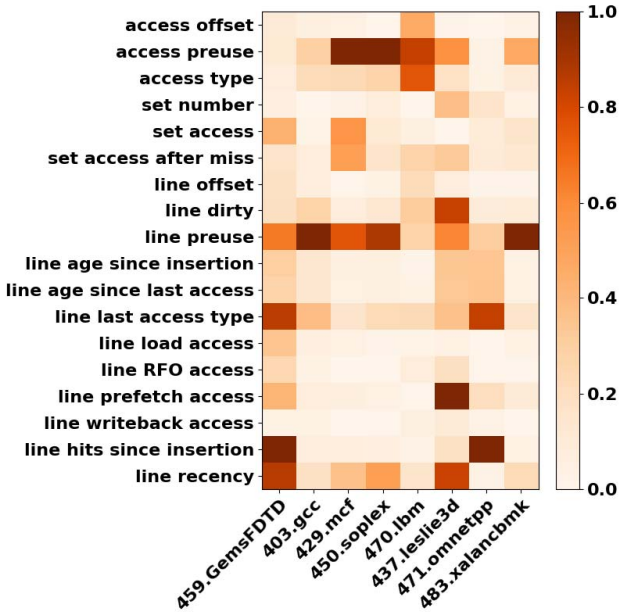

Fig. 3. Heat map of neural network weights. The y-axis shows features representing LLC state, and the x-axis shows the benchmarks used in the agent simulation. The features with high magnitude of weights are (considering at least three benchmarks) access preuse, line preuse, line last access type, line hits since insertion, and line recency.

To achieve performance similar to that of a neural network while avoiding the associated hardware overhead, we analyze and draw insights from a trained neural network to derive a practically implementable replacement policy.

Applications that show significant difference in LLC hit rates between Belady and LRU replacement policies were chosen for testing. Although state-of-the-art replacement policies perform better than LRU in these applications, there is still a performance gap between these policies and Belady, which provides scope for us to improve our policy. We use reinforcement learning to guide us through the process. We allow a neural network to explore paths unexplored by the contemporary replacement algorithms with the target of closing the performance gap between Belady and other replacement policies. After training, we analyze neural network weights for all selected benchmarks. As stated in Section III-A, on a LLC miss, the LLC state representing features of missed access and the accessed set is sent as input to the agent. Then, the agent provides a replacement decision to be followed. To comprehend important features in the LLC's state that affect the agent's decision, we compute the average weight of each individual input layer neuron over all neurons in the hidden layer. For cache line features, we also compute the average across all cache ways in a set. Figure 3 shows the heat map of feature weights such that the higher and lower magnitude weights are depicted at different ends of the color spectrum.

Typically, a feature is more important if it has higher magnitude weights (darker color in the heap map). Although the heat map can help identify important features for making good replacement decisions, it is left to us to understand why

these features are important and how they impact replacement decisions. Ultimately, we would want to utilize these features and derive a practical replacement policy. Implementing the agent's neural network directly in hardware is unlikely practical.

The first challenge in the process of deriving our own replacement policy is to minimize the search space and focus only on critical features. We use hill-climbing analysis together with machine learning to finalize our target feature set. We started by training the agent with only one feature at a time. After doing this for each individual feature, we select the feature that performs the best. Then we enable this feature with one additional feature and evaluate all such feature pairs. We repeat the process by adding one more feature at a time until no further performance improvement is seen. This hill-climbing analysis yields a set of five features.

In this section, we define each of these features and discuss the insights that we derive from them. Later, we design a new policy based on these features that can be implemented in hardware with acceptable overheads.

1) Preuse Distance: We define preuse distance as the past reuse distance of a cache line. It is computed as the number of set accesses between the last access and the current access of the cache line. Based on the heat map, both access preuse and line preuse features show high magnitude of weights.

Access Preuse is the preuse distance of the cache line that is accessed by the current request. To obtain the preuse distance for every cache access, one must keep a record of all previously accessed addresses, refer to the record when serving a new access, and compute its preuse distance accordingly. Although we implement the record keeping and lookup function in our simulation framework, doing so in hardware can be very costly. As a result, we do not consider this feature for our final policy.

Line Preuse refers to the preuse distance of a cache line. To compute line preuse, we add counters for every cache line. On a set access, the counters of all cache lines in that set are incremented. If the access is a hit, the counter value corresponds to its preuse distance. On a miss, a new cache line is installed. We then reset the counter and start counting the preuse distance of the newly inserted cache line. In Section IV, we propose optimizations to reduce the counter overhead.

How does preuse distance contribute to the policy that the agent learned? Recall in Section III-A that the agent tries to learn the behavior of Belady optimal policy, i.e., replace the cache line with the farthest reuse distance. However, reuse distance is not provided as an input feature. Our conjecture is that preuse distance is related to reuse distance in certain scenarios. During a program execution, the distance between two accesses of an address could be constant. For example, the number of memory accesses in each iteration of a for loop can be the same. In such a scenario, if the same address is accessed in every iteration, its reuse distance will be constant. However, this may not be true in the case of an LLC access because of the filtering effect of private caches. In addition, prefetch and writeback accesses can impact reuse distance.

To comprehend the relationship between preuse and reuse distance at LLC, we analyze the difference between preuse and reuse distance for every LLC access. Figure 4 shows the percentage of reused cache lines with absolute difference between preuse and reuse distance below 10 (i.e., $|preuse\ distance - reuse\ distance| < 10$), between 10 and 50, and greater than 50. For a significant number of cache lines, we can approximate the reuse distance using preuse distance, as the difference between preuse and reuse distance is less than 10 accesses. For more than 50% of the reused cache lines, the difference between preuse and reuse distance is less than 50 accesses. To allow these cache lines to be reused, we can retain the cache lines for a few more accesses after their preuse distance has been reached. We should also note that Figure 4 shows the absolute difference between preuse and reuse. This means that the reuse distance could be smaller than the preuse distance, so some cache lines might be reused before reaching their respective preuse distances.

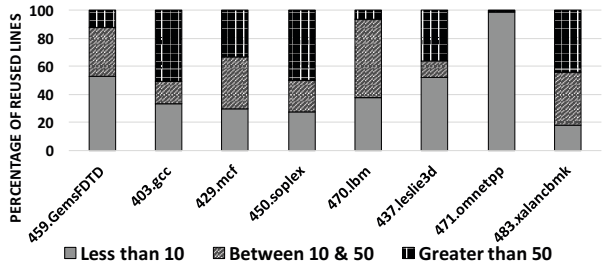

Fig. 4. Difference between preuse and reuse distance for reused cache lines.

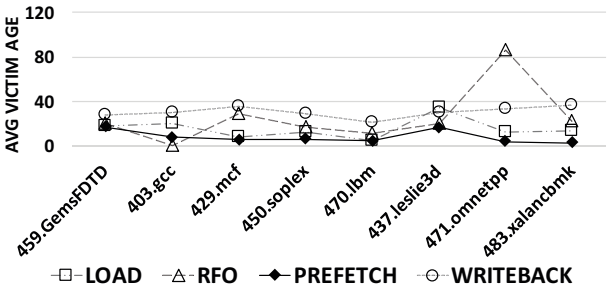

Fig. 5. Average victim age for each access type.

2) Line Last Access Type: Last Access type of a cache line is defined as the latest access' type. To understand its significance, Figure 5 shows the average victim age for each access type. We accumulate the age since the last access for each victim chosen by the RL agent and compute the average for each access type. In almost all benchmarks, prefetch access has the lowest average victim age. This implies that the prefetched cache lines have the lowest cache life time, and the agent prefers to evict them sooner than the cache lines from other access types. However, prefetched cache lines contribute to significant number of demand hits for a few benchmarks, like 459.GemsFDTD, 437.leslie3d, and 429.mcf. Therefore, we infer that the reuse distance of prefetched cache lines is small, and it suffices to have a short cache life time for prefetched cache lines. This ensures that non-reused prefetched cache lines are evicted sooner, allowing other cache lines to be reused.

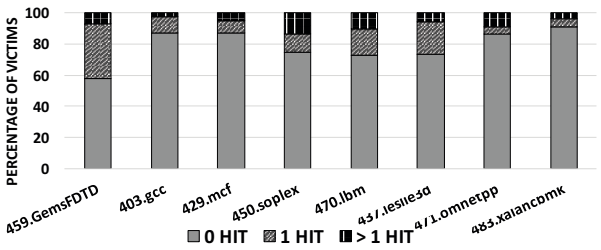

Fig. 6. Number of hits when a cache line is evicted.

3) Line Hits Since Insertion: Hits since insertion tells us how many times a cache line has been accessed since it was brought into the cache. To understand its significance, Figure 6 shows the percentage of victim cache lines that are evicted with zero, one, and more than one hits. In all benchmarks, more than 50% of victims have zero hits, and more than 80% of victims have at most one hit. The insight from this analysis is that the agent tends to evict cache lines with fewer hits. When designing a cache replacement policy, we can mimic this behavior by retaining cache lines that have more hits.

4) Recency: Recency refers to the relative access order of a cache line in a set. Recency value ranges from zero to (*Set Associativity* − 1); zero indicates the least recently used cache line, and (*Set Associativity* − 1) indicates the most recently used cache line. For example, LRU replacement policy replaces the cache line with recency value 0.

To understand the significance of recency, we plot the percentage of victims evicted by the agent, segregated by recency of the victims, in Figure 7. We observe that most evictions occur with cache lines with a high recency value, implying that the agent prefers to evict cache lines that are most recently used. To comprehend this behavior, note that the agent is rewarded positively for evicting cache lines that are either not reused or reused later than the other cache lines in the set. When the agent evicts a cache line with a high recency value, it means that the older cache lines (recency value close to 0) are reused before the newer cache lines (recency value close to 15). For example, when two cache lines in the set have the same reuse distance, the older cache line will be reused before the newer cache line. So, the agent chooses the newer cache line for eviction. Given the high percentage of victims with high recency values in agent simulations, we take the insight that evicting more recently accessed cache lines has a better chance of maximizing demand hits.

C. Summary

In this section, we presented one viable way to draw insights from an ML model for cache replacement policies. First, we identify important features by analyzing the weights of the agent neural network. Next, we try to understand the behavior of each feature by looking into its relevant statistics collected from architecture simulations. Through this ML-based analysis, we benefit from the following:

1) *Reducing exploration time*: Several cache replacement policies are built on heuristics identified from common access patterns. New heuristics can be derived through creative and aggressive design of experiments for the cache replacement problem; however, this process is time consuming and limited by a designer's imagination. RL lets the agent perform the heavy lifting by running simulations using different input features. For input features that perform well, we analyze the neural network and decipher the agent's replacement policy.

2) *Dynamically adapting policy*: The RL agent adapts to dynamic changes in access stream and handles non-trivial consequences of chosen actions. The agent's dynamic policy works well for all benchmarks. Analyzing the policy helps us capture this dynamism in a cost-effective replacement policy.

3) *Rigorously confirming the importance and sufficiency of heuristically-known features*: Though we considered non-obvious features like line address offset, set number, set accesses after miss, etc., our ML model picked features that heuristically have been known to be useful, such as reuse distance and hits since insertion. In addition to identifying features through the heat map analysis, we use hill-climbing analysis to select a set of the most critical features, as described in Section III-B. We also perform the analysis on benchmarks that show significant difference in LLC hit rate between Belady and LRU replacement policies. This rigorous approach proves the importance of selected features.

4) *New perspective on using features in a replacement policy*: By analyzing agent simulations, we identified a different approach for using some features in our replacement policy. For example, rather than segregating cache lines into clean and dirty, we use a cache line's access type to categorize it as prefetched or not. This allows us to predict whether the cache line will hit in the future, as described in Section III-B2.

5) *Automation of feature selection*: The entire process of feature selection, from agent simulation, neural network weight analysis, to hill climbing analysis was automated. Although in this paper we use heat map analysis for visual convenience, the weight comparison and feature count reduction were automated. Through this work, we show that ML is an effective tool for tackling challenging computer architecture design problems. An automated ML-based cache replacement policy can match or beat state-of-the-art hand-crafted designs.

For the cache replacement problem targeted in this work, we have the following insights. 1) Preuse distance can be used to estimate reuse distance of a cache line, which is essential for making good replacement decisions. This insight is drawn from the *line preuse* feature. 2) Cache lines loaded by prefetch accesses are reused within short time intervals. This insight is drawn from the *line last access type* feature. An efficient cache replacement policy can use this insight to evict non-reused prefetched cache lines. 3) A cache line that has been accessed multiple times is likely to be accessed again. This insight is drawn from the *line hits since insertion* feature. 5) Sometimes it is beneficial to evict the youngest cache line. This insight is drawn from the *line recency* feature.

IV. Reinforcement Learned Replacement (RLR)

We propose a replacement policy (RLR) based on insights learned from the neural network. At a high level, RLR follows the following rules.

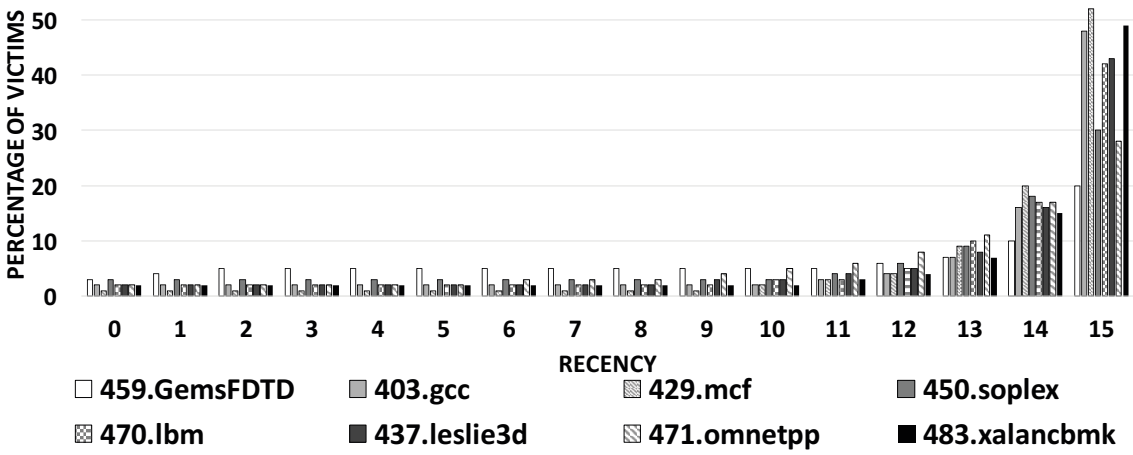

Fig. 7. Recency for victims in Agent simulation.

1) For a significant number of cache lines, the reuse distance can be approximated by preuse distance.
2) The type of previous access of a cache line can be used to predict its chance of receiving a hit.
3) Cache lines that have been accessed can be predicted to be accessed again in the future.
4) Recently-inserted cache lines are prioritized for eviction to allow older cache lines to be reused.

A. Replacement Algorithm

In RLR, reuse distance is predicted based on the preuse distance of the cache lines. Cache lines with age less than the predicted reuse distance are protected. In addition, cache lines are prioritized based on the type of their previous access and whether or not the cache lines have received hits. When a replacement decision is made, the cache lines in the set are assigned priority levels. Priority levels are computed based on the cache line's age, previous access, and hits. On a cache miss, the cache line with the lowest priority will be evicted.

Age priority (P_{age}): Each cache line is augmented with an *Age Counter* that counts the line's age in set accesses (i.e., how many times the set has been accessed since the last access of the line). On a demand hit, the counter's value represents the preuse distance of the accessed cache line. Because we use preuse distance to approximate future reuse distance (Section III-B1), we predict that the cache line will be reused after a number of set accesses equal to the preuse distance. If the cache line is not accessed after the predicted reuse distance, it is considered for eviction. However, maintaining registers to store predicted reuse distance for each individual cache line is impractical. Instead, we accumulate the preuse distances of cache lines on demand hits and use the accumulated value to approximate future reuse distance (RD). Cache lines are protected from eviction until their respective ages reach RD. RD must be chosen carefully. On the one hand, if RD is too high, cache lines with small reuse distance might be retained in the cache longer than necessary. On the other hand, if RD is too low, cache lines with large reuse distance might be evicted prematurely before reuse. Also, reuse distance changes during application execution. Therefore, RD must be updated periodically to adapt to application phases. In our experiment, RD is updated for every 32 demand hits, by averaging the aggregated preuse distance and then multiplying by two (i.e., $RD = 2 \times Average\ Preuse\ Distance$). Recall that for most cache lines, the preuse distance and reuse distance are not exactly the same. Doubling the average preuse distance can be beneficial because it allows cache lines to stay in the cache longer. Overall, age-based priority levels are assigned as follows.

- **Priority Level 1:** If Age Counter is smaller than RD, the cache line has not yet reached the reuse distance; this cache line might be reused in the future. Higher priority is assigned to retain the cache line for future reuse.
- **Priority Level 0:** If Age Counter is greater than RD, the cache line has already reached the reuse distance, but has not yet been reused since last access. Lower priority is assigned because the line might not be reused in the future.

Type priority (P_{type}): Each cache line is augmented with a *Type Register*, indicating whether its previous access was a prefetch. Type-based priority levels are assigned as follows.

- **Priority Level 1:** If the last access type is not prefetch, then either the cache line was not inserted by a prefetch access, or it has been reused after insertion. We want to keep this cache line.
- **Priority Level 0:** If the last access type is prefetch, it has not been reused. When replacement is needed, we tend to evict non-reused prefetched cache lines.

Hit priority (P_{hit}): Each cache line is augmented with a *Hit Register* that is set when the cache line is hit. The hit-based priority levels are assigned as follows.

- **Priority Level 1:** If the Hit Register is 1, the cache line has been reused. We want to keep this cache line because the data can be accessed repeatedly in the same program.
- **Priority Level 0:** If the Hit Register is 0, the cache line has not been reused. When replacement is needed, we tend

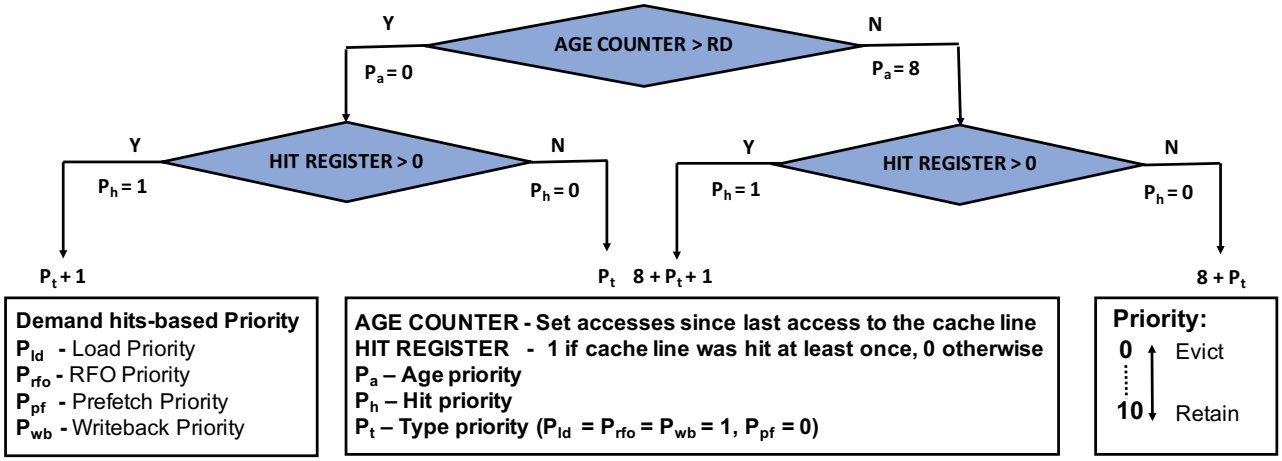

Fig. 8. Flowchart for priority computation in RLR.

to evict non-reused cache lines.

The priority level for each cache line is computed as a weighted sum of the priorities described above, given by the following equation.

$$P_{line} = 8 \cdot P_{age} + P_{type} + P_{hit}$$

The weights are designed based on hill climbing analysis, described in Section III-B. Agent performance was analyzed by enabling one feature and disabling the rest. Among all the features, preuse distance achieved the highest performance. Therefore, we assign it the highest weight. The features last access type and age contributed equally. Figure 8 shows the flowchart for priority computation in RLR. The age priority of a cache line is computed by comparing its Age Counter with RD. If the cache line's age is greater than RD, age priority (P_{age}) is set to 0, otherwise it is set to 8. The value 8 (for P_{age}) is chosen for two reasons. First, we give higher weights to cache lines whose age is less than RD, with the hope that they can be retained in the cache for a longer period and reused in the near future. Second, multiply by 8 can be implemented in hardware efficiently by left shifting three times. The hit priority P_{hit} of the cache line is computed from the Hit Register. P_{hit} is 1 if the Hit Register is set; otherwise it is 0. The type priority P_{type} of the cache line is computed from the Type Register. P_{type} is 0 if the Type Register indicates a prefetch access; otherwise it is 1.

To select a replacement candidate, the cache management policy selects the way with the lowest priority level. It is possible that multiple ways have the same priority level. To break ties, we use the recency information.

Recency: Each cache line has a $log(Set\ Associativity)$-bit value indicating the relative order of access among the lines in a set. When multiple lines have the same priority, the most recently accessed cache line (high recency value) is evicted. The most recently accessed cache line takes the longest time to reach the RD value. Evicting it allows the other cache lines to reach the RD value. If cache bypass is supported, the cache management policy bypasses a request if no cache line has reached an age greater than the RD value. In RLR, recency plays an important role in victim selection. However, tracking recency accurately for every cache line can be costly. In Section IV-C, we describe an optimization technique to represent recency using fewer bits.

B. Hardware Implementation

Each cache line is equipped with an Age Counter, a Hit Regsiter, and a Type Register. The Age Counter is an n-bit saturating counter, tracking 2^n number of set accesses. When a demand hit occurs, the cache line's Age Counter is sent to the Accumulator. After the number of demand hits reaches a threshold (32 in this case), we average the accumulated value and then double the value. The averaging circuit can be as simple as a right shifting logic, as long as the demand hit threshold is a power of 2. For example, to average over $2^5 = 32$ cache hits, the accumulator value is right shift by 5. Also, the averaged preuse distance can be doubled by left shifting 1 bit. We combine the averaging and doubling circuit by right shifting the accumulated value by $(5-1) = 4$ bits.

The hardware implementation for computing RD is shown in Figure 9. The Hit Register of a cache line is set when it receives a hit. A Type Register is used to indicate if the cache line's previous access type was prefetch or not. To estimate the hardware cost of RLR, we synthesize the design in 32 nm technology using Synopsys Design Compiler [26]. The area, power, and latency for RLR are 84μm^2, 4.6μW, and 0.68ns, respectively. The cache line priority computation in RLR can be done in parallel to the tag comparison (to determine hit/miss) and does not contribute to the critical path latency. RD computation in RLR is done when LLC is idle or fetching data from memory, so that it is not part of the critical path. The area overhead is negligible compared to total processor area. For example, the area of Intel Sandy Bridge processor with similar configuration is 216 mm^2 [10].

C. Optimizations

To determine the Age Counter's optimal size, we ran simulations by varying the number of Age Counter bits from 2 to 8 bits per cache line. To achieve good performance while keeping the overhead low, we chose 5 bits to represent Age Counter. We verified that 5 bits are sufficient to cover

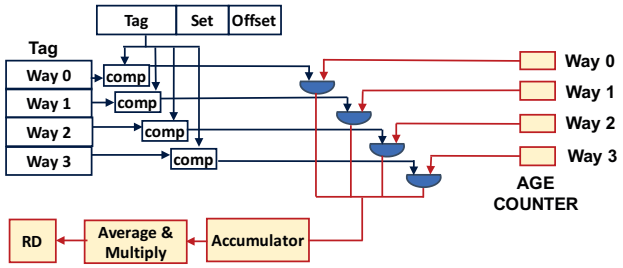

Fig. 9. Hardware implementation for computing RD. On a demand hit, the cache line's age value is sent to the RD computation circuit.

the average preuse distance in agent simulations for most benchmarks. In addition, we used a 1-bit Hit Register, a 1-bit Type Register, and $log(Set\ Associativity)$ bits for recency of a cache line. In a 16-way set associative cache, this amounts to 11 bits of overhead per cache line. To further reduce overhead, we devised two optimizations.

Age Counter Optimization: There are two ways to minimize the overhead of the Age Counter – counting fewer events and approximating counter value. To count fewer events, we use Age Counter to count the number of set misses instead of set accesses. After a hit, cache lines in a set remain unchanged. By counting set misses, Age Counter represents the relative age of a cache line (since its last access) rather than the absolute age. To approximate counter value, we increment Age Counter for every 8 set misses. This allows us to reduce the overhead per line by 3 bits. We use a 3-bit counter per set to count every set miss. After 8 set misses, the counter rolls over and the line counters are incremented.

Recency Approximation: We can use the age of a cache line to determine its recency. The most recently accessed cache line is either hit or inserted in the cache. In both cases, the age of the line is zero. For a hit, the age counter value is sent to the accumulator for computing RD, then reset. On a miss, a new line replaces a victim, and the corresponding age counter is reset. Therefore, the most recently accessed line can be identified by age counter value zero. In a 16-way set associative cache, using age counter to determine recency reduces the overhead by 4 bits per cache line. In RLR, recency is used to break ties when multiple cache lines have the same priority level. When multiple cache lines have the same age counter value and the lowest priority level, we chose to evict the cache line with the lowest way index.

In summary, after optimization, each cache line has a 2-bit Age Counter, a 1-bit Hit Register, and a 1-bit Type Register, totaling 4 bits of overhead per cache line. In addition, we use a 3-bit counter per set. For a 2MB 16-way LLC with 64B cache line, the total storage overhead of RLR is 16.75KB.

D. Multicore Extension

In a multicore system that executes different benchmarks on separate cores, cache lines in the LLC can be segregated based on the ids of the cores that issue the accessed requests. Although each core (benchmark) has a different reuse characteristic, it is challenging to predict their behavior when accesses from all cores are mixed. This is because two consecutive accesses from the same core can be interleaved by multiple accesses from other cores. However, we observe that the access frequency of a core and its average reuse distance have an inverse correlation. That is, a core that has the most number of LLC accesses within a time interval also has the smallest average reuse distance. This is because a cache line from a core having high access frequency tends to be reused earlier than a cache line from a core with low access frequency. This information can be used in the replacement policy to select a victim with large reuse distance by selecting a cache line from a core with low access frequency.

In RLR, we assign priorities for each core. When a replacement decision is made, each cache line in the set is assigned a priority based on its age, hit, type and core. The cache line with the lowest priority is chosen for eviction. Based on our experiments, assigning core priorities based on demand hit frequency instead of access frequency yields better performance. For this, we maintain demand hit counters for each core at the LLC. On every demand hit (Load or RFO hit), the LLC demand hit counter corresponding to the core of the cache line is incremented. Based on the number of demand hits, each core is assigned a **Priority Level (0, 1, 2, or 3)**. A core with more demand hits is assigned a higher priority level. The core priorities (P_{core}) are updated for every 2000 LLC accesses. In terms of overhead, the demand hit counters contribute 12 bits per core to the overall storage overhead. The priority level for each cache line (P_{line}) in a set is computed by the following formula.

$$P_{line} = 8 \cdot P_{age} + P_{type} + P_{hit} + P_{core}$$

TABLE III
PARAMETERS FOR THE EVALUATED SYSTEM

Core	6-stage pipeline, 3-issue O3, 256-entry ROB
L1 I-Cache	32 KB, 8-way, 4-cycle latency, LRU
L1 D-Cache	32 KB, 8-way, 4-cycle latency, LRU
L2 Cache	256 KB, 8-way, 12-cycle latency, LRU
LLC (per core)	2 MB, 16-way,26-cycle latency
Prefetcher	next-line (L1), IP-stride (L2), None (LLC)

V. EVALUATION

A. Methodology

We implement RLR in ChampSim simulator from the 2^{nd} Cache Replacement Championship (CRC2). We estimate the performance on 1-core and 4-core configurations with a 6-stage pipeline and a 256-entry reorder buffer. The memory system has a 3-level cache hierarchy with private L1, L2 and a shared LLC. The complete configuration is shown in Table III. We use SPEC CPU® 2006 [2] and CloudSuite [8] benchmarks for evaluation. To train the RL agent, we only used the first 100M instructions of eight SPEC CPU benchmarks. In evaluation, however, we also show results for 26 new benchmarks that have not been used in training.

For SPEC CPU 2006 evaluation, we use all 1 billion instruction traces from SimPoint [1] provided by CRC2. For Cloudsuite benchmarks, we use all traces files provided by CRC2. We warm the cache for 200 million instructions and evaluate the performance for the next one billion instructions. In 4-core simulations, we evaluate performance when four different benchmarks are run simultaneously on separate cores.

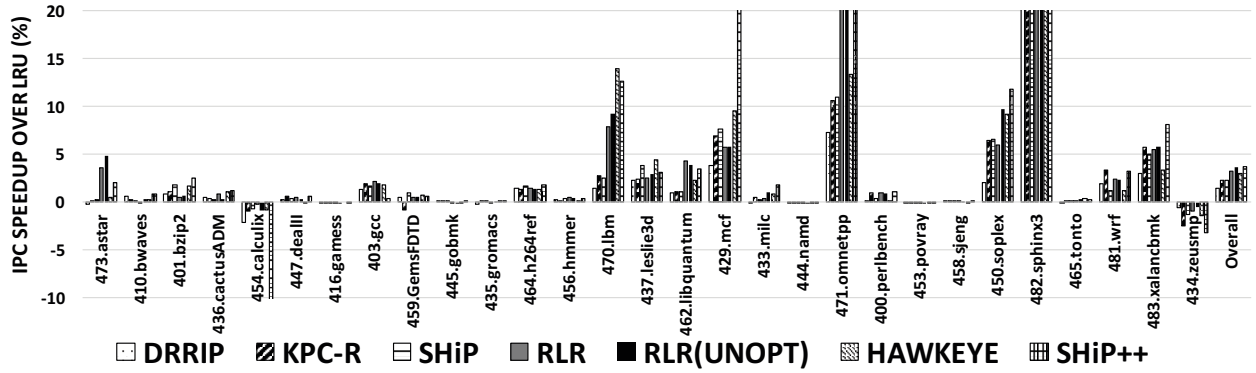

Fig. 10. Performance comparison for different LLC replacement policies (SPEC2006).

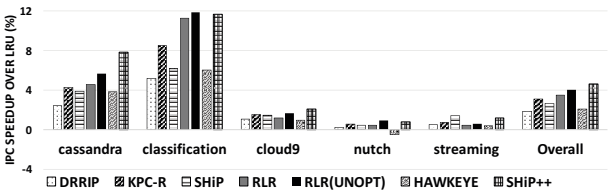

Fig. 11. Performance comparison for different policies (Cloudsuite).

We generate 100 random sets of four benchmarks from the 29 applications in SPEC CPU 2006. We use the same trace files as in single-core simulations. However, we run the simulation until each benchmark completes one billion instructions. If any benchmark reaches the end of its trace, the corresponding core continues simulating from the beginning of the trace file. In 4-core simulations for Cloudsuite benchmarks, we run each trace in its respective core. For the results of single-core simulations, we use IPC speedup over LRU. The IPC speedup of each benchmark i is measured as $\frac{IPC_i}{IPC_{i,LRU}}$. If a benchmark has more than one trace file, we present the geometric mean of all IPC speedup numbers. For multicore results, the overall performance of each workload mix is measured as the geometric mean of IPC speedups of all benchmarks in the mix $(\prod_{i=1}^{4} \frac{IPC_i}{IPC_{i,LRU}})^{\frac{1}{4}}$.

B. Experimental Results

In this section, we compare the performance of RLR against KPC-R, DRRIP, and SHiP, as well as policies from CRC2, including SHiP++ and Hawkeye. We obtained the source code for these policies from the CRC2 website. We also compare against some prior works such as EVA [4] and PDP [6].[2] Compared to LRU, we observed IPC degradation in both EVA and PDP. For SPEC CPU 2006, EVA degrades single-core system performance by 0.11%, and PDP degrades performance by 3.72%, on average. The original works on EVA [4] and PDP [6] show performance improvements with respect to LRU and DRRIP. Considering that the margin of difference in performance between any two LLC replacement policies is small, the selection of instruction traces used for evaluation can have significant impact on overall results. Using a rigorous evaluation methodology is important. Performance discrepancies for EVA and PDP may be attributed to use of single instruction traces, not based on SimPoints [1], that do not fully characterize an entire application. We observed that overall ranking of policies can change between individual SimPoints. As such, we compute results from all SimPoints to ensure accurate representation of benchmark behavior.

In addition to results for overhead-optimized policies, we also present the results of RLR without overhead reduction optimizations (Section IV-C), denoted by RLR(unopt). In the unoptimized policy, we use a 2-bit Hit Counter, as opposed to a 1-bit Hit Register. Each cache line has a 5-bit Age Counter, a 2-bit Hit Counter, and a 1-bit Type Register, amounting to 10-bit overhead per cache line. For a 2MB 16-way LLC, the total storage overhead is 40KB for the unoptimized policy.

Figures 10 and 11 show performance comparisons for SPEC CPU 2006 and Cloudsuite benchmarks, respectively. RLR outperforms KPC-R and DRRIP for all benchmarks. For our evaluations, we used the IP-stride prefetching policy at L2. Since the performance of KPC-R is improved by information from KPC-P prefetching, we also compared KPC-R and RLR with KPC-P as the L2 prefetching policy. In such a system, KPC-R and RLR improve performance by 3.9% and 5.5%, respectively, for SPEC CPU 2006. For Cloudsuite, KPC-R and RLR improve performance by 2.46% and 3.5%, respectively. RLR performs better than KPC-R by evicting non-reused prefetched cache lines in LLC sooner. KPC-P avoids cache pollution in two ways. First, low-confidence prefetches are not inserted in L2. Second, when a prefetch access hits in LLC, the corresponding cache line is promoted in the replacement stack only when the prefetch confidence is higher than a threshold. While the first method prevents cache pollution in L2, the second method does not evict non-reused prefetched cache lines in LLC sooner than cache lines from other access types. RLR also outperforms SHiP for most benchmarks. For example, in 470.lbm, prefetching does not improve IPC significantly. RLR evicts prefetched cache lines sooner than other cache lines, resulting in better performance compared to SHiP. In memory-intensive benchmarks such as 429.mcf, SHiP maximizes hit rate by ranking PCs in the order of hit contribution and retaining cache lines fetched by highly

[2]We procured EVA source code from http://people.csail.mit.edu/sanchez. We implemented PDP as described in [6].

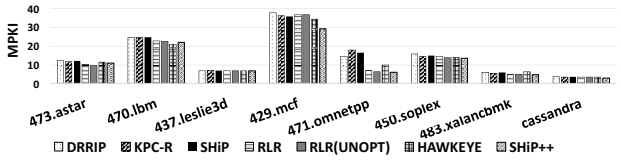

Fig. 12. Demand MPKI comparison for different policies.

TABLE IV
OVERALL SPEEDUP FOR DIFFERENT REPLACEMENT POLICIES.

Policy	1-core (2MB LLC)		4-core (8MB LLC)	
	SPEC 2006	Cloud-Suite	SPEC 2006	Cloud-Suite
DRRIP	1.50 %	1.80 %	2.63 %	1.07 %
KPC-R	2.30 %	3.07 %	5.50 %	3.80 %
RLR	3.25 %	3.48 %	4.86 %	2.39 %
RLR(unopt)	3.60 %	4.02 %	5.87 %	2.5 %
SHiP	2.24 %	2.64 %	6.33 %	3.09 %
Hawkeye	3.03 %	2.09 %	7.69 %	2.45 %
SHiP++	3.76 %	4.60 %	7.37 %	3.89 %

ranked PCs. For a few benchmarks, like 437.leslie3d and streaming (Cloudsuite), RLR has a larger number of LLC demand hits (Load and RFO) compared to SHiP. However, the trend does not reflect in IPC. This is because SHiP is more likely to retain cache lines from PCs contributing to IPC improvement, while RLR retains cache lines contributing to demand hits.

Table IV summarizes performance improvement over LRU for evaluated replacement policies for single-core and multicore workloads. For single-core and multicore evaluation, overall performance improvement is computed as the geometric mean of IPC speedup for all evaluated workloads. In single-core evaluations, RLR outperforms DRRIP, KPC-R, SHiP (PC-based), and Hawkeye (PC-based) by 1.74%, 0.95%, 1.01%, and 0.22%, respectively, for SPEC CPU 2006. In Cloudsuite, RLR outperforms DRRIP, KPC-R, SHiP (PC-based), and Hawkeye (PC-based) by 1.65%, 0.41%, 0.84%, and 1.39%, respectively. Figure 12 shows Misses Per Kilo-Instructions (MPKI) for benchmarks with MPKI greater than 3. Compared to DRRIP, RLR achieves a maximum of 52% reduction in 471.omnetpp and a minimum of 2.5% in 429.mcf.

We simulated policy variants by eliminating hit and type priorities to evaluate their contributions to RLR's performance. In SPEC CPU 2006, IPC speedup over LRU reduces by 12% when the hit register is disabled. This shows that protecting cache lines that received at least one hit over the cache lines that were never hit has a significant impact on performance. When the type register is disabled, speedup reduces by 30%, demonstrating that significant performance gains are achieved by protecting cache lines from one access type over another.

In multicore evaluation (Figure 13), RLR outperforms DRRIP by 2.3% and 1.32% in SPEC2006 and Cloudsuite, respectively. Contrary to the single-core results, KPC-R outperforms RLR by 0.64% and 1.41% in SPEC2006 and Cloudsuite, respectively. This is because interference from other core accesses delays the reuse of prefetched cache lines. However, RLR allows prefetched cache lines to be reused within short time intervals. RLR performance can be improved

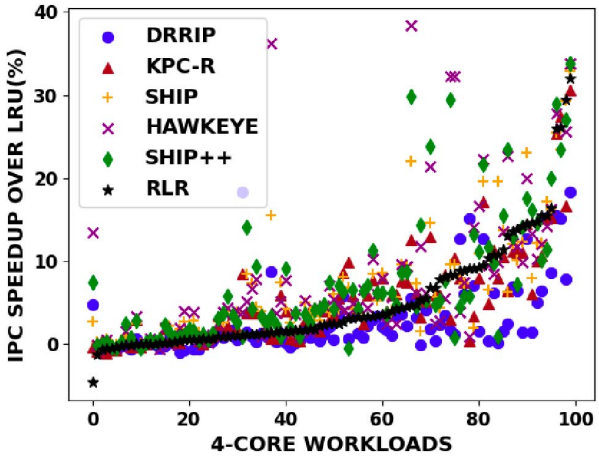

Fig. 13. Performance comparison of different policies in the 4-core setup.

by considering each core's access frequency for the eviction of prefetched cache lines. SHiP also performs better than RLR by 1.32% and 0.7% in SPEC2006 and Cloudsuite, respectively. Analyzing the workloads in which SHiP outperforms RLR reveals that in some workloads, a small percentage of PCs account for nearly all demand hits. This application characteristic allows any PC-based policy to protect cache lines frequently accessed by those small percentage of PCs while evicting cache lines brought in by other PCs. Given the large number of LLC accesses in a multicore system, the information brought by PCs from different cores is useful. Though we lose some information by avoiding PC usage, RLR captures the benchmarks characteristics through features that can be computed readily at the LLC and achieves a performance similar to the PC-based policies. Using KPC-P instead of IP-stride prefetching policy, RLR outperforms KPC-R by 0.5%, indicating that without using PC in the memory system, RLR performs better than other non-PC based replacement policies. Also, with RLR, we avoid the complexity of designing and verifying a multicore system that incorporates hardware infrastructure for accessing PC at LLC. We have the luxury of building a standalone cache design that can be integrated with already designed and verified single/multi-core systems.

VI. CONCLUSION

Machine learning is useful in architecture design exploration. However, human expertise is still essential in deciphering the ML model, making design trade-offs, and finding practical solutions. In this work, with the goal of designing a cost-effective cache replacement policy, we used reinforcement learning to guide and expedite our design. We trained an RL agent using features that are relatively easy to obtain at the LLC. Considering the complexity in propagating PC information to the LLC, we intentionally excluded PC from the feature set. After training the agent neural network, we identified important features from a large feature set by analyzing neural network weights. Based on insights drawn from the neural network, we successfully derived a new

replacement policy. We then optimized the proposed policy to further reduce hardware overhead. Overall, the proposed replacement policy outperforms DRRIP (non-PC-based policy) and achieves comparable performance to existing PC-based replacement policies.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback. Authors would also like to thank Yasuko Eckert and Gabriel Loh for their feedback that helped guide this work in its early stages.

REFERENCES

[1] "SimPoint," http://www.cs.ucsd.edu/users/calder/simpoint/.
[2] "SPEC CPU 2006," https://www.spec.org/cpu2006/.
[3] "The 2nd cache replacement championship," 2017. [Online]. Available: https://crc2.ece.tamu.edu/
[4] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *HPCA*, 2017.
[5] S. Das, T. M. Aamodt, and W. J. Dally, "Reuse distance-based probabilistic cache replacement," *TACO*, vol. 12, no. 4, 2015.
[6] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *MICRO*, 2012.
[7] V. V. Fedorov, S. Qiu, A. N. Reddy, and P. V. Gratz, "Ari: Adaptive llc-memory traffic management," *TACO*, vol. 10, no. 4, 2013.
[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *ACM SIGPLAN Notices*, vol. 47, no. 4, 2012.
[9] Q. Fettes, M. Clark, R. Bunescu, A. Karanth, and A. Louri, "Dynamic voltage and frequency scaling in nocs with supervised and reinforcement learning techniques," *IEEE Transactions on Computers*, vol. 68, no. 3, 2018.
[10] Henry, *intel 32nm-22nm comparison*. [Online]. Available: http://blog.stuffedcow.net/2012/10/intel32nm-22nm-core-i5-comparison
[11] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *ISCA*, 2016.
[12] 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, 2010.
[13] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *HPCA*, 2001.
[14] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *MICRO*, 2017.
[15] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *ICCD*, 2007.
[16] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutluy, and D. A. Jimenezz, "Improving cache performance using read-write partitioning," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 452–463.
[17] S. M. Khan, Y. Tian, and D. A. Jiménez, "Dead block replacement and bypass with a sampling predictor," in *MICRO*, 2010.
[18] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, 2008.
[19] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," 2017.
[20] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *MICRO*, 2008.
[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.
[22] S. optimizing memory controllers: A reinforcementlearning approach, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
[23] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007.
[24] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *MICRO*, 2019.
[25] R. Sutton and A. Barto, *Reinforcement Learning*. MIT Press, 1998.
[26] Synopsys, *Design Compiler User Guide*. [Online]. Available: http://www.synopsys.com/
[27] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches," in *Supercomputing*, 2004.
[28] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *MICRO*, 2016.
[29] G. Tesauro, "Online resource allocation using decompositional reinforcement learning," in *AAAI*, 2005.
[30] 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 *MICRO*, 2011.
[31] C.-J. Wu and M. Martonosi, "Adaptive timekeeping replacement: Fine-grained capacity management for shared cmp caches," *TACO*, vol. 8, no. 1, 2011.
[32] J. Yin, Y. Eckert, S. Che, M. Oskin, and G. H. Loh, "Toward more efficient noc arbitration: A deep reinforcement learning approach," *AIDArch 2018*.
[33] J. Yin, S. Sethumurugan, Y. Eckert, C. Patel, A. Smith, E. Morton, M. Oskin, N. E. Jerger, and G. H. Loh, "Experiences with ml-driven design: A noc case study," in *HPCA*, 2020.
[34] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *CRC*, 2017.
[35] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher: a case study," in *MEMSYS*, 2017.
[36] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *DAC*, 2019.