# ReSemble: Reinforced Ensemble Framework for Data Prefetching


Pengmiao Zhang
*University of Southern California*
Los Angeles, USA
pengmiao@usc.edu

Rajgopal Kannan
*United States Army Research Laboratory*
Los Angeles, USA
rajgopal.kannan.civ@army.mil

Ajitesh Srivastava
*University of Southern California*
Los Angeles, USA
ajiteshs@usc.edu

Anant V. Nori
*Intel Corporation*
Bangalore, India
anant.v.nori@intel.com

Viktor K. Prasanna
*University of Southern California*
Los Angeles, USA
prasanna@usc.edu



***Abstract*—Data prefetching hides memory latency by predicting and loading necessary data into cache beforehand. Most prefetchers in the literature are efficient for specific memory address patterns thereby restricting their utility to specialized applications–they do not perform well on hybrid applications with multifarious access patterns. Therefore we propose ReSemble: a Reinforcement Learning (RL) based adaptive enSemble framework that enables multiple prefetchers to complement each other on hybrid applications. Our RL trained ensemble controller takes prefetch suggestions from all prefetchers as input, selects the best suggestion dynamically, and learns online toward getting higher cumulative rewards, which are collected from prefetch hits/misses. Our ensemble framework using a simple multilayer perceptron as the controller achieves on the average 85.27% (accuracy) and 44.22% (coverage), leading to 31.02% IPC improvement, which outperforms state-of-the-art individual prefetchers by 8.35%–26.11%, while also outperforming SBP, a state-of-the-art (non-RL) ensemble prefetcher by 5.69%.**



***Index Terms*—reinforcement learning, ensemble, prefetching**


## I. Introduction

Memory latency is a major bottleneck in computer performance [1], [2], more so given recent advances in AI accelerators and data intensive workloads. Hardware prefetching is an effective way to hide memory latency and improve IPC (instructions per cycle). A prefetcher anticipates future cache misses and fetches data from the memory hierarchy before a processor requests the data [3]. Existing prefetchers [4]–[21], usually exploiting spatial or temporal localities [22], efficient for specific access patterns but cannot adapt to multiple applications with various patterns. Crucially, different applications may benefit from different prefetchers at different phases (see Section II).

Ensemble prefetching can overcome the limitations of single prefetchers. Existing ensemble prefetching methods rely on the classification of memory access patterns [23]–[25] or the evaluation of recent history performance among multiple prefetchers [26], [27]. However, both methods have important limitations. Classification methods perform *offline training* of models, thereby assuming consistency between offline training and online testing memory access patterns. In fact, training memory traces are usually sampled rather than inclusive, which can cause low accuracy when a trained model encounters inconsistent patterns. In addition, classification methods ignore the *interaction* between the prefetcher and the caches, which leads to low adaptability in dynamic scenarios. In contrast, performance evaluation methods, such as Sandbox Prefetcher (SBP) [26], involve online cache feedback but are limited by *response lag*: a picked prefetcher works for a period until the average performance of another prefetcher surpasses it. The sub-optimal prefetcher has worked for a number of accesses before being replaced, which can lead to low prefetching performance for interleaving patterns.

Our goal is to train an ensemble prefetching controller that dynamically selects the prefetching suggestion from multiple prefetchers. The controller should be able to 1) train and update online, 2) learn from the interaction between prefetcher and cache, and 3) respond at the level of each access instead of an access sequence (thereby avoiding response lag).

Reinforcement learning (RL) [28] is a machine learning technique that enables an intelligent agent to learn optimal actions that maximize a cumulative reward by interacting with the environment. While RL for prefetching has been explored in the literature, prior work has been limited to designing *single prefetchers* based on heuristic locality assumptions, such as spatial range [29] or semantic locality [30]. In contrast, we propose to use RL for *ensemble prefetching*. Compared with supervised and unsupervised machine learning algorithms [31], RL fits better to our problem due to several advantages: 1) There is no need for offline training–an RL agent continuously learns online. 2) By defining the ensemble controller as the RL agent and the memory hierarchy equipped with multiple prefetchers as the environment, an ensemble controller can interact with the memory and learn from cache feedback, i.e., prefetch hit/miss. 3) The RL agent takes action by observing the current environment state instead of performance history, which can lead to accurate and quick responses.

Applying RL to ensemble prefetching is challenging. The first challenge is to design the input for the ensemble

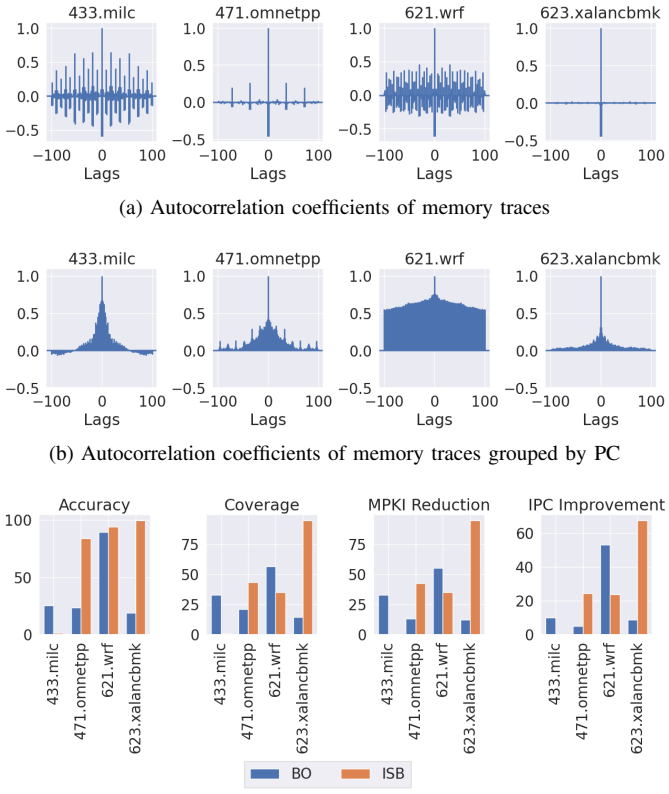

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

prefetcher (known as *observation* in the RL context). Typical applications of RL, such as robotics and Atari games in OpenAI [32], use image pixel data as observation. However, there is no such straightforward observation for an ensemble prefetching controller. Second, the memory address space is both vast and sparse – memory traces can contain millions of unique addresses under 32 or 64 bit addressing. Whether viewed as numerical values or classes, this creates computational challenges in processing (known as the *class explosion* problem in prefetching [33]). Third, the cache feedback for a given prefetch is not instantaneous. The ensemble controller has to wait a number of accesses to know if a prefetch is useful, i.e. the prefetch is requested before being replaced. Therefore, an asynchronous learning scheme is essential.

In this paper, we propose ReSemble, a Q-learning [34], [35] based RL framework to train an ensemble prefetching controller. We develop three novel methods to overcome the challenges described above. First, we define the prefetching predictions from multiple sources as the *observation* from the controller. Second, to address the *class explosion* problem, we use hash functions to preprocess the observation and generate state vectors, which can serve as indexes if using tabular models. However, the state space can still be quite large for a table-based predictor straining storage resources. Therefore, we apply *hash and norm* for state vector generation, and use a simple and compact multilayer perceptron (MLP) as the Q-network, which outputs the selection of the best prefetching suggestion. Third, to address the lag of feedback, we propose a *lazy sampling* mechanism that separates the collection of state transitions and the rewards feedback, sampling only from the rewarded transitions for model training. Furthermore, for the ease of hardware implementation, we also develop a tabular variant of ReSemble that uses a table-based controller to select the best prediction and uses tokenization to compress the model size. Using both state-of-the-art individual and ensemble prefetchers as baselines, we evaluate our method through simulation experiments.

We summarize our main contributions below:

- We propose ReSemble, an ensemble prefetching framework based on reinforcement learning that takes prediction from multiple prefetchers as input and selects the best prediction driven by rewards from future prefetch hits/misses. It uses *hash and norm* for preprocessing to solve the class explosion problem in memory address, uses MLP as the ensemble controller to make prefetching decisions, and uses *lazy sampling* mechanism for online training to address the lag of cache feedback.
- We present a compact tabular variant of ReSemble, using a simple table-based controller to select the best prediction. We reduce the address space using hash functions and reduce table size using tokenization, which largely compresses the model and makes it feasible for hardware implementation.
- We evaluate the learning performance of ReSemble and the tabular variant. Results show that ReSemble with an MLP-based controller achieves higher average rewards, better adaptability, and quicker response with a smaller model size compared with the tabular variant.
- We conduct a comprehensive comparison among ReSemble, its tabular variant, individual prefetchers, and a state-of-the-art ensemble prefetcher SBP (Sandbox prefetcher). Results show that the MLP-based ReSemble achieves an average of 85.27% in accuracy and 44.22% in coverage, leading to 31.02% IPC improvement, outperforms state-of-the-art individual prefetchers by 8.35%–26.11%, outperforms SBP by 5.69%. Tabular variant achieves 83.94% accuracy, 42.15% coverage, and 29.26% IPC improvement, outperforming individual prefetchers by 6.59%–24.35% and outperforming SBP by 3.93%.




(a) Autocorrelation coefficients of memory traces

(b) Autocorrelation coefficients of memory traces grouped by PC

(c) Performance of different prefetchers

Fig. 1: Different patterns of memory traces and the performance improvement gained from different prefetchers.

## II. Motivation

In this section, we analyze the memory trace patterns of several applications and show how different prefetchers perform differently, which motivates us to develop ensemble methods for higher prefetching performance. We use SimPoint [36] to generate the memory miss traces under 20 million instruction of the last level cache (LLC) from four applications: *433.milc* and *471.wrf* from benchmark SPEC 2006 [37]; *621.wrf* and *623.xalancbmk* from benchmark SPEC 2017 [38].

Figure 1a shows the autocorrelation plots for the generated traces. An autocorrelation plot reveals the periodicity or repeating patterns of a time-series data sequence by computing

the autocorrelation coefficients (ACs) at varying time lags. From Figure 1a we can observe that *433.milc* shows significant positive spikes at various lags that is evidence of autocorrelation. The decay of ACs indicates the higher independence of memory accesses to longer distance history. While *621.wrf* also shows high positive spikes, it reveals longer lag dependencies from the observation of slower decay of ACs. The autocorrelation of *433.milc* and *621.wrf* indicates that they are amenable to prefetchers detecting memory access deltas, e.g. spatial prefetchers BO and SPP. In contrast, *471.omnetpp* and *623.xalancbmk* show insignificant spikes of ACs, indicating low periodicity. Therefore, the trace patterns are difficult to learn and temporal prefetchers may perform better.

Figure 1b shows the autocorrelation plots for the traces that are grouped by PC (program counters). We group the memory accesses by PC while keeping the access order within each PC. In this way, we explore whether the traces show different ACs getting rid of the influence of interleaved PCs. *433.milc* shows a faster decay of ACs after removing the interleaved PCs, which means patterns per PC can reduce its long-distance dependence. *471.omnetpp*, *621.wrf*, and *623.xalancbmk* all show increased ACs after the traces are grouped by PC, which indicates that PC information can help a prefetcher to better detect trace patterns. Most notably, the PC grouped *471.omnetpp* and *623.xalancbmk* show significant autocorrelations with small lags comparing to the original traces. This means that prefetchers tracking the sequence per PC tend to perform better for these two applications.

Figure 1c shows the performance of two prefetchers, BO [6] and ISB [8], applied to the applications discussed above. BO is a spatial prefetcher that scores the deltas in an offset list by checking whether the delta has made a hit in recently requested accesses. The prediction of BO is constrained within a page. ISB is a temporal prefetcher that tracks the memory access sequence per PC and predicts the future accesses by replaying the recorded memory access sequences. We can observe how the features of BO and ISB influence their performance in Figure 1c. BO provides higher coverage, MPKI (miss per kilo instructions) reduction and IPC improvement for application *433.milc* and *621.wrf*, and shows low contribution to the performance improvement of *471.omnetpp* and *623.xalancbmk*, which matches our analysis regarding the autocorrelation coefficients. In contrast, ISB contributes more performance improvement for *471.omnetpp* and *623.xalancbmk*. Notably, ISB shows higher accuracy when applied to *621.wrf* though it shows lower coverage and contributes less to the application performance improvement, which demonstrates the characteristic of temporal prefetchers.

The analysis above demonstrates that it is difficult for a single prefetcher to outperform other prefetchers dealing with various memory access sequences. Consequently, we seek to leverage the benefits from multiple prefetchers and use a general and adaptable ensemble framework to select from the prefetching suggestions predicted from multiple prefetchers.

## III. Background

### A. A Taxonomy of Hardware Prefetchers

Hardware prefetching techniques have been evolved from simple rule-based stride and stream prefetching to adaptive table-based methods learning from memory address correlations. There are different taxonomies for prefetchers. In the concern of pattern localities [39], prefetchers can be classified as spatial prefetchers, temporal prefetchers, and spatio-temporal prefetchers [40]–[42], as is shown in Table I. Different memory access sequences benefit the best from different prefetchers as is illustrated in Figure 2.

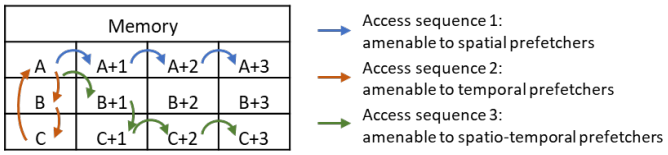



Fig. 2: Types of memory access sequences and prefetchers.

**Spatial prefetchers** rely on the *spatial locality* of memory reference streams, i.e. the property that an access to a memory location indicates that a physically nearby location will be accessed with high probability in the near future [39]. *Spatial locality* gives insight that prefetching strategies can perform well even focusing only on a small fixed-size section, referred to as *Spatial Regions* [42], typically the size of a page. Spatial prefetchers such as BO [6], VLDP [7], and SPP [43], learn from history access page offsets or access deltas, then predict future accesses within a fixed spatial region. Though spatial prefetchers work effectively for programs with spatial localities, such as list traversing, they can not track long-distance dependencies beyond the spatial region, such as traversing a large graph that nodes are stored in multiple pages.

**Temporal prefetchers** rely on the *temporal locality* of memory reference streams, which means that the same location as the current memory access will very likely be accessed again in the near future [39]. A typical strategy for temporal prefetchers, such as STMS [12], ISB [8], and Donimo [14], is to record a history access sequences and replay the sequence as predictions when there is a match. For the above temporal prefetchers, while STMS and Donimo track global temporal patterns, ISB exploits the temporal locality for each program counter (PC). Though PC information benefits patterns like loop and self increments, constraining sequences for one instruction can damage the global patterns. Shortcomings of the temporal prefetching strategy are notable. Temporal prefetchers highly rely on the repetitive appearance of sequences and cannot provide a reasonable prediction for compulsory misses.

**Spatio-temporal prefetchers** try to utilize both spatial and temporal localities. The most well-known spatio-temporal prefetcher is STeMS [9]. STeMS records the temporal history accesses for each PC as well as the access offsets within the located spatial region. When a sequence trigger is encountered, the prefetcher reconstructs the memory access sequence using stored offsets and deltas based on spatial locality. It has been

TABLE I: A Taxonomy of Prefetchers Based on Address Correlations

| Types | Examples | General Mechanisms | Main Shortcomings |
|---|---|---|---|
| Spatial prefetcher | BO, VLDP, SPP | Predict offsets within a spatial region | Long distance dependencies |
| Temporal prefetcher | ISB, STMS, Domino | Rrecord and replay history misses in order | Compulsory misses |
| Spatio-temporal prefetcher | STeMS | Capture temporal patterns and predict misses within a spatial region | Low coverage and high start-up latency |

proved that the performance of STeMS varies dramatically for different applications. It suffers from low prefetching coverage and high start-up latency, making STeMS ineffective [42]. Due to the high complexity and low efficiency, few recent prefetchers were developed under this track.

### B. Reinforcement Learning and Q-Function

Reinforcement Learning (RL) concerns how intelligent agents ought to take actions in an environment in order to maximize the cumulative reward [44], which is typically formulated as a Markov Decision Process (MDP) [45] with unknown dynamics [46]. An MDP is defined as a five-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$ [47]; $\mathcal{S}$ is the state space of an agent; $\mathcal{A}$ is the agent's action space; $\mathcal{T}(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ defines the transition dynamics, which returns the probability that the agent transits from state $s$ to $s'$ by taking action $a$; $\mathcal{R}(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the reward function that defines the intermediate reward received when the agent transits from state $s$ to $s'$ by taking action $a$; $\gamma \in [0, 1)$ determines the reward discount factor per time-step, which decreases the importance of distant future rewards. The objective of reinforcement learning is to find a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ such that the expected future discounted reward is maximized. Formally, a Q-function, $Q^{\pi} : S \times A \rightarrow \mathbb{R}$, is defined to express the expected future discounted reward as:

$$Q^{\pi}(s, a) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{t} \mathcal{R}(s_t, a_t, s_{t+1}) \mid s_0 = s, a_0 = a, \pi\right] \tag{1}$$

According to the Bellman equation [45], the Q-function for the optimal policy $Q^*$ can be expressed recursively as:

$$Q^*(s, a) = \sum_{s' \in S} \mathcal{T}(s, a, s') \left[\mathcal{R}(s, a, s') + \gamma \max_{a'} Q^*(s', a')\right] \tag{2}$$

Given $Q^*$, the optimal policy $\pi^*$ can be recovered by selecting the action that provides the highest Q-value:

$$\pi^*(s) = \arg\max_{a} Q^*(s, a) \tag{3}$$

A variety of learning algorithms seek to estimate $Q^*$ for recovering the optimal policy $\pi^*$ for an agent. Q-Learning [34] and Deep Q-Network (DQN) [35] are two notable algorithms.

**Q-Learning** iteratively improve its approximation to the Q-function by update a Q table.

While the tabular implementation of Q-function is simple for implementation, it only works in environment with discrete, finite, and small state space and action space. Practically, for extremely large input space, like memory addresses space, the tabular Q-learning is incapable to process directly. A memory trace can have millions of unique addresses, whose permutation as table index can be very storage consuming.

**Deep Q-Network** uses a convolutional neural network parameterized by weight $\theta$ to approximation the Q-function. The network parameters are updated by gradient descent. DQN can handle large state space because it inferences through computation instead of table look up.

To make the Q network feasible for hardware, we use a *shallow Q-network*, a simple multilayer perceptron (MLP) with one hidden layer, to approximate the Q-function and handle the large input space. Besides, we also develop a feasible tabular Q-learning implementation with a simple structure. We show that MLP-based model achieves better performance compared to tabular model while using less space.

## IV. Our Approach: ReSemble

In this section we describe ReSemble, our ensemble framework for adaptively leveraging the predictions from multiple prefetchers. Figure 3 shows the integration of ReSemble into hardware architecture for last level cache (LLC) prefetching. Multiple prefetchers read memory accesses of LLC and provide suggestions to an ensemble controller. We formulate ensemble prefetching as a reinforcement learning problem. The learning relies on interactions between the *environment* and an *agent*. As is shown in Figure 4, in ReSemble:

- *environment* is the hardware architecture, including the memory hierarchy, the equipped multiple prefetchers, the peripheral interfaces, and the accessory hardware for prefetching;
- *agent* is the ensemble controller that aims to dynamically manage the selection of prefetching suggestions from multiple sources.

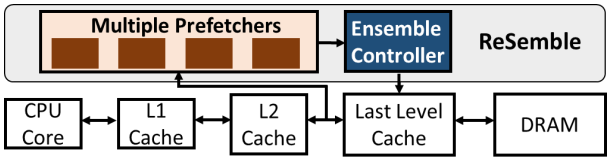



Fig. 3: Integrating ReSemble in hardware architecture.

**Problem Formulation.** Given states $\mathcal{S}$ acquired from an architecture with multiple prefetchers working in parallel (*environment*), we aim to train an ensemble controller (*agent*) using samples of state transitions $\mathcal{T}$ that selects actions from set $\mathcal{A}$ to maximize a long term reward $\mathcal{R}$ that reflects the effectiveness of prefetches at a discount factor of $\gamma$.

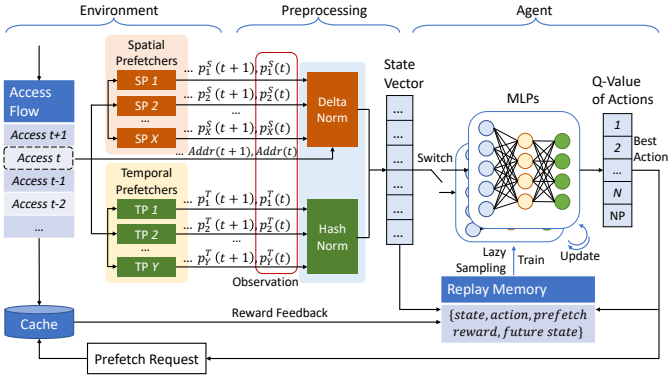



Fig. 4: Overall design and workflow of ReSemble. Through interactions with the environment (architecture with multiple prefetchers), an MPL-based agent (ensemble controller) is trained to maximize the long term reward, which reflects the effectiveness of prefetches.

### A. Overall Design and Workflow

Figure 4 shows the overview of ReSemble. First, multiple prefetchers provide predicted addresses respectively according to their own prefetching strategies. The predicted addresses from prefetchers are preprocessed to a state vector (see Section IV-B). Then, an MLP-based ensemble agent is designed to approximate the Q-function, which can select the best action given the current state vector (see Section IV-C). Meanwhile, a replay memory is designed to collect state transitions consisted of {*state, action, prefetch, reward, future state*} for model training (see Section IV-D1), in which the reward is collected from future prefetch hits/misses (see Section IV-D2). Due to the asynchronism of action and reward, the transitions are sampled using a *lazy sampling* mechanism for MLP training (see Section IV-D3). Using the sampled transitions, the MLP-based agent can be trained online, the training algorithm is presented in Section IV-E. In addition, a tabular variant is provided in Section IV-F with a simpler structure for the ease of hardware implementation.

### B. Preprocessing

The agent takes the memory address predictions from prefetchers as observation from the environment. We classify the prefetcher predictions to *spatial predictions* that are typically within a spatial region predicted by spatial prefetchers, denoted as $p_n^S(t)$, and *temporal predictions* that are in the range of full memory address space and typically predicted by temporal prefetchers, denoted as $p_n^T(t)$. The classification is based on the output range rather than the constraints of prefetcher taxonomy considering the emerging and future prefetchers using hybrid address correlations. Formally, we define the observation at time $t$ from the environment as $o_t$:

$$\begin{aligned} o_t &= [p_1(t), p_2(t), ...p_N(t)] \\ &= [p_1^S(t), ..., p_X^S(t), p_1^T(t), ..., p_Y^T(t)] \end{aligned} \tag{4}$$

where $N$ is the total predictions at time $t$, $X$ is the number of spatial predictions, and $Y$ is the number of temporal predictions, $N = X + Y$. For the situation of no prediction or variant degrees for multiple prefetchers, the observation uses zero padding to fill and keep the size of the observation vector.

We need to preprocess observation $o_t$ to state vector $s_t$ that can be used as the input of neural network, the state vector $s_t$ is defined as:

$$\begin{aligned} s_t &= f_{prep}(o_t) \\ &= [s_1^S(t), ..., s_X^S(t), s_1^T(t), ..., s_Y^T(t)] \end{aligned} \tag{5}$$

where $f_{prep}$ is the preprocessing function that maps elements in observation vectors to elements in state vectors.

We design different preprocessing methods for the two types of observations, $p_n^S$ and $p_n^T$, as is shown in Equation 6.

**Spatial predictions preprocessing.** The spatial predictions are within a range so we use the deltas, the difference between the predicted address $p_n^S(t)$ to the current address $Addr(t)$, for model input. In practice, spatial prefetchers usually predict deltas first then add to the current address. Thus, we simply remove their post-processing step and output the deltas. We use the spatial range $2^{PAGE\_BITS}$ to normalize the input.

**Temporal predictions preprocessing.** We use a *hash and norm* approach to handle the large address space from temporal predictions $p_n^T(t)$. We use a $HASH\_BITS$ bit length folding method as the hash function to compress the absolute predicted address. To normalize the result, the hashed values are divided by $2^{HASH\_BITS}$.

$$\begin{aligned} s_n^S(t) &= \frac{|p_n^S(t) - Addr(t)|}{2^{PAGE\_BITS}} \\ s_n^T(t) &= \frac{hash(p_n^T(t))}{2^{HASH\_BITS}} \end{aligned} \tag{6}$$

There are several reasons that the observations are not appropriate to be directly used as neural network inputs. For one thing, due to the extremely large address space, the input value can be very large and cannot be effectively processed by MLP. For another, spatial prefetchers only predict in a very small range while temporal prefetchers predict accesses over the whole address space. Thus, larger input values would dominate the learning and inference. Tokenizing the addresses is a commonly used method to address the above problem [17], [33], [48], [49], but it requires extra storage for millions of mappings. Therefore, we design the hash and norm method for preprocessing. Through a combination usage of delta, hashing, and normalization, we convert observation vectors to state vectors.

### C. Multilayer Perceptron

We use multilayer perceptron (MLP) networks to parameterize the Q-function in Equation 2. An MLP is a feedforward artificial neural network that typically consists of only three layers of nodes: an input layer, a hidden layer, and an output layer. We choose MLP because of its simple and compact structure that increases the feasibility of potential hardware implementation. An MLP in the ensemble structure takes the state vector as input and output the predicted Q-values of the individual actions for the input state:

$$Q(s_t, a_t) = MLP(s_t, a_t; \theta) \tag{7}$$

where $a_t \in \mathcal{A} = \{1, 2, ..., n, ..., N, \text{NP}\}$ is the specific action at time step $t$, $n$ is the index of the predicted address $p_n(t)$, and NP means no prefetching.

To make a decision of action taking, we apply decaying $\epsilon$-greedy algorithm that balances the exploration and exploitation trade-off. The action at time $t$ is determined by:

$$a_t = \begin{cases} \text{Random selection from } \mathcal{A} & \text{if } P < \varepsilon \\ \text{argmax}_{a \in \mathcal{A}} MLP(s_t, a_t; \theta) & \text{otherwise} \end{cases} \quad (8)$$

where $P$ is a probability, $\epsilon$ decays from $\epsilon_{start}$ toward $\epsilon_{end}$ with the number of *steps* and a *decay* factor: $\epsilon = \epsilon_{start} + (\epsilon_{start} - \epsilon_{end}) e^{-\frac{\text{step}}{decay}}$. The decaying $\epsilon$ enables the model to adapt fast to the memory access pattern.

### D. Data Collection and Sampling

*1) Replay memory:* We implement a replay memory to store the last $N$ tuples of transitions $(s_t, a_t, p_t, r_t, s_{t+1})$, which form a dataset $\mathcal{D}$ with transitions {*current state, action, prefetch, reward, future state*}. By sampling from $\mathcal{D}$, the stored transitions can be reused for model training.

*2) Reward Feedback:* Since the Q-function is optimized to acquire the highest long-term reward, we need to define the reward in a way to indicate the prefetching effectiveness. For a coming access, the requested address $Addr$ is compared to the history prefetching addresses $p_t$ within a history window $W$. If there is a match at time $i$ for $p_i$, the prefetching hits and the corresponding reward will be set to $r_i = 1$, otherwise, if there is no hit and the prefetching is beyond the history window $W$, the reward will be set as $r_i = -1$. The above scenario works when the agent prefetches valid addresses. If the agent action is NP, the reward can be set directly as 0 when the transition tuple is pushed into the replay memory.

*3) Lazy sampling:* Considering the lag of cache feedback, we develop a *lazy sampling* mechanism for online training. At time step $t$, the current state $s_t$ and $a_t$ can be immediately stored in the replay memory. At $t+1$, the state vector $s_{t+1}$ updates the transition $\mathcal{T}_t$ stored in $D$ at time step $t$ and fill in the valid $s_{t+1}$ value. For future reward feedback, the current prefetching address $p_t$ is also stored along with $s_t$ and $a_t$. Then using the reward rule in Section IV-D2, update $r_t$ in $\mathcal{T}_t$ where there is a valid reward feedback. As a result, only the transitions with valid reward can be sampled for model training, and invalid transitions will be pended. This process is referred to as *lazy sampling*.

### E. Online Training

Algorithm 1 shows the complete process of the ReSemble inference and online training. To approximate the Q-function, two MLP networks are defined: one policy network denoted as $MLP_p$, and one target network denoted as $MLP_t$. The policy network $MLP_p$ is used for online training for each $I_p$ steps. $I_p$ is small, which enables quick adaptation to the observations from the environment. Instead, the target network $MLP_t$ updates by loading weights from $MLP_p$ for each $I_t$ steps, where $I_t > I_p$. $MLP_t$ works for the inference of action selection, the approximation of the future rewards, and the learning objective of $MLP_p$. The training data for $MLP_p$ is randomly selected using lazy sampling from the transitions with valid rewards, $(s_t, a_t, p_t, r_t, s_{t+1})$, in the replay memory $D$. The loss function for optimizing the $MLP_p$ is:

$$L(\theta) = \mathbb{E}_{s,a \sim \rho(s,a)} \left[ (y_i - MLP_p(s, a; \theta))^2 \right] \quad (9)$$

$$y_j = r_j + \gamma \max_{a'} MLP_t(s_{j+1}, a'; \theta') \quad (10)$$

where $\rho(s, a)$ is a probability distribution over sequence $s$ and action $a$, $y_j$ is the expected accumulative reward of next step, which is inferenced from the target network $MLP_t$.

**Algorithm 1** ReSemble Inference and Online Training

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize policy net $MLP_p$ with random weights $\theta$
Initialize target net $MLP_t$ with weights $\theta' = \theta$
Initialize state vector $s_0$
Initialize transition vector $\mathcal{T}_t = (s_t, a_t, p_t, r_t, s_{t+1})$
Configure window $W$
Configure $MLP_p$ update interval $I_p$ and $MLP_t$ update interval $I_t$, where $I_p < I_t$
**for** $t = 1, T$ **do**
    $s_t \leftarrow f_{prep}(o_t)$
    **if** $P < \epsilon$ **then** ▷ Decaying $\epsilon$-greedy
        Select a random action $a_t$ from $\mathcal{A}$
    **else**:
        Select $a_t = \max_a MLP_t(s_t, a; \theta)$ ▷ Inference
    **end if**
    **if** $a_t$ is NP **then**
        Execute no operation
        Set reward $r_t \leftarrow 0$
    **else**
        Prefetch predicted address $p[a_t]$
    **end if**
    Update $(s_t, a_t, p_t)$ in $\mathcal{T}_t$ ▷ Lazy sampling
    Push transition $\mathcal{T}_t$ to replay memory $\mathcal{D}$
    Update transition $\mathcal{T}_{t-1}$ with $s_t$ in $\mathcal{D}$
    **for** prefetching $p_i$ in $\mathcal{D}$ within window $W$ **do**
        **if** current address $Addr = p_i$ **then**
            $r_i \leftarrow 1$ for $\mathcal{T}_i$ within window $W$
        **end if**
    **end for**
    $r_i \leftarrow -1$ for $\mathcal{T}_i$ beyond $W$ without valid reward
    Label $\mathcal{T}_i$ as valid
    **if** $kI_p$ steps **then** ▷ Policy net online training
        Sample valid transitions from $\mathcal{D}$
        $y_j \leftarrow r_j + \gamma \max_{a'} MLP_t(s_{j+1}, a'; \theta')$
        $\theta \leftarrow \theta + \alpha (y_j - MLP_p(\cdot; \theta)) \nabla_\theta MLP_p(\cdot; \theta)$
    **end if**
    **if** $kI_t$ steps **then**
        Swap $(MLP_t, MLP_p)$ ▷ Nets role switch
        $\theta' \leftarrow \theta$ ▷ Target net updating
    **end if**
**end for**

Through one step of gradient descent on the loss function, the parameters $\theta$ in the policy network $MLP_p$ can be updated as:

$$\theta \leftarrow \theta + \alpha \left(y_j - MLP_p(s_j, a_j; \theta)\right) \nabla_\theta MLP_p(s_j, a_j; \theta) \quad (11)$$

After each $I_t$ steps, the target net $MLP_t$ will be updated by loading the parameters from the policy net $MLP_p$. To avoid pending, we propose using a simple switch to swap the role of the $MLP_t$ and $MLP_p$. That is, when the policy net is trained for $I_t$ steps, the policy net will be switched on as the network for inference, serve as the target net for the next $I_t$ steps, while the target net will be switched off from inference, and will be used for training as policy net for next $I_t$ steps. The networks synchronization can start after a quick switch, which avoids the latency of model weight loading.

### F. Tabular Variant

For ease of hardware implementation, we develop a simple tabular RL ensemble controller based on Q-learning, which uses a Q-table to approximate the Q-function in Equation 2. The challenge of tabular implementation is the vast state space, which causes large Q-table size when using the states as table index. We address this challenge by two steps of compression: address compression using hash function and Q-table compression using tokenization.

First, we use folding method hash function for both spatial and temporal predictions without the step of normalization, as is shown in Equation 12. Using n-bit hash function can map the address space to the range of [0, $2^n$).

$$\begin{aligned} s_n^S(t) &= hash(p_n^S(t) - Addr(t)) \\ s_n^T(t) &= hash(p_n^T(t)) \end{aligned} \quad (12)$$

Though the address space has been reduced by hashing, the theoretical storage is still very large if directly using state vector as index: $2^{(nS)}A$ for state dimension $S$ and action space $A$. We observe that the state space is sparse. Therefore, we tokenize the unique states and use the tokens instead of the state vectors as the index to implement the Q-table.

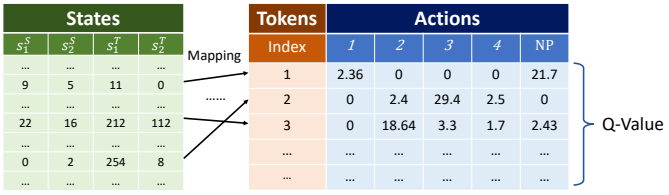



Fig. 5: Q-table for a tabular variant of ReSemble, using 8-bit hash function to reduce address space and tokenization to reduce table size.

Figure 5 shows the structure of the proposed Q-table in the case of four source prefetchers. Given a state, the vector will be mapped to a corresponding Q-table index. The Q-values show how good each actions are for a certain state. Q-value is updated online for each time step following Equation 13.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max Q'\left(s', \cdot\right) - Q(s,a)\right] \quad (13)$$

where $Q(s,a)$ is the Q-value for the pair of state-action, $r$ is the reward for taking action $a$, $s'$ is the next state after taking action $a$. Considering the lag of reward we apply the same lazy sampling method similar as in Section IV-D. The Q-value is updated only when a valid reward is available. Instead of a replay memory, A small buffer to store pending transitions is enough for the tabular variant, because the table use one transition for a step of update instead of a batch of transitions.

## V. Evaluation

ReSemble is a versatile framework that is open to architectures equipped with various numbers and types of prefetchers. In this section, we implement an example framework for evaluation, and compare the performance to state-of-the-art individual and ensemble prefetchers.

### A. Methodology

*1) Configuration:* We use four prefetchers as ReSemble input, the budget of the input prefetchers are shown in Table II, including two spatial prefetchers and two temporal prefetchers:

- **Best-offset (BO) prefetcher** [6] is a spatial prefetcher. BO algorithm tries to find the optimal prefetching offset by testing a list of deltas.
- **Signature Path Prefetcher (SPP)** [43] is also a spatial prefetcher but it is able to detect when a data access pattern crosses a page boundary. It also uses path confidence to balance the aggressiveness of prefetching.
- **Irregular Stream Buffer (ISB) prefetcher** [8] is a temporal prefetcher that learns temporally correlated memory accesses based on PC-localized stream.
- **Domino prefetcher (Domino)** [14] is another temporal prefetcher that concerns about the effectiveness prefetching, using only the history of both one and two last miss addresses to find a match for prefetching.

TABLE II: Configuration of Input Prefetchers

| Prefetchers | Configuration | Budget |
|---|---|---|
| BO [6] | 1K entry RR table, 1Kb prefetch bits | 4KB |
| SPP [43] | 256 entry ST, 512 entry PT,<br>1024 entry PF, 8 entry GHR | 5.3KB |
| ISB [8] | 2K entries for SP-AMC and PS-AMC | 8KB |
| Domino [14] | 2KB prefetch buffer, 256B PointBuf,<br>128B LogMiss, 64B FetchBuf | 2.4KB |

The framework configuration is shown in Table III. The left column of Table III shows the parameters of the environment and the preprocessing, which is determined by the architecture configuration. The right column shows the hyperparameters for training the agent and are acquired from grid search. They can keep fixed if only replacing input prefetchers while keeping the input dimension (number of input prefetchers).

We implemented ReSemble based on MLP and its tabular variants based on 4-bit and 8-bit hashing. The comparison of model size is shown in Table IV, where $H$ is the hidden layer dimension of MLP, $B$ is number of bits used in hashing. In MLP implementation, the model size is $SH + HA + H + A$ considering both the model weight and bias. For table-based

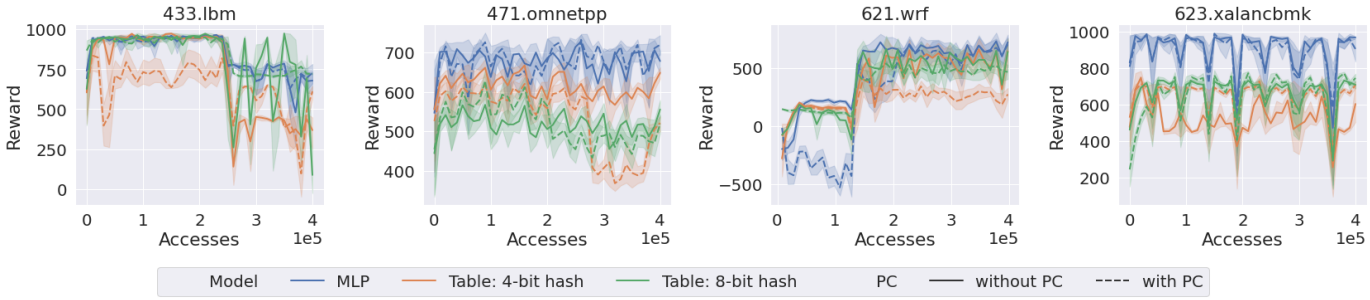
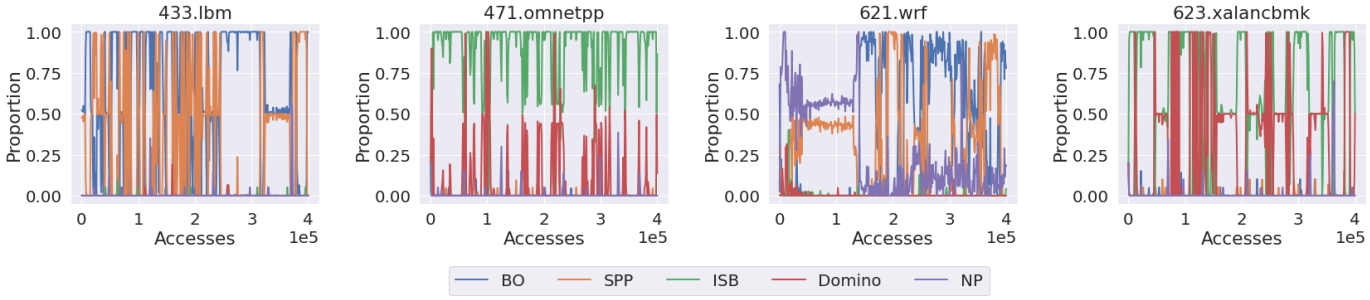
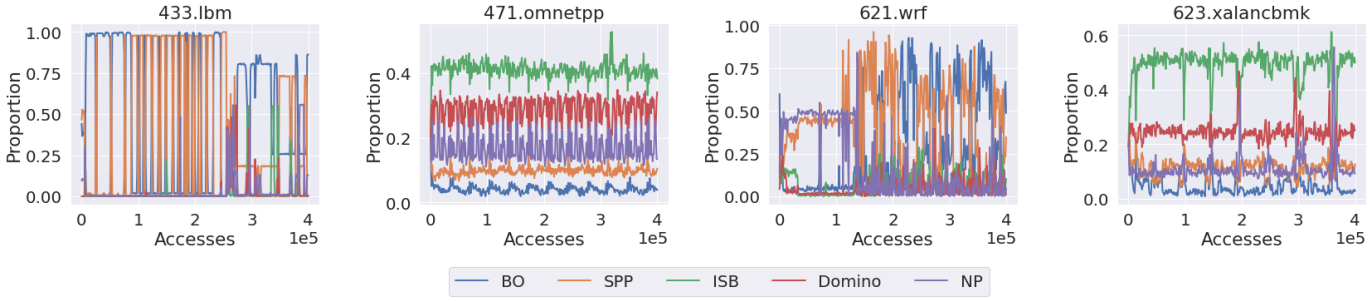



Fig. 6: Case study on the learning process of MLP-based and tabular RL ensemble controllers.




(a) Actions of MLP-based RL ensemble controller.




(b) Actions of 8-bit tabular RL ensemble controller.

Fig. 7: Case study on the actions of MLP-based and tabular RL ensemble controllers.

TABLE III: Configuration of ReSemble Framework

| **Configuration (env)** | **Info** | **Configuration (agent)** | **Info** |
|---|---|---|---|
| Address bit | 64 | Replay memory $R$ | 2000 |
| Block offset | 6 | Prefetch window size $W$ | 256 |
| Page offset | 12 | Batch size for training | 256 |
| State dimension $S$ | 4 | $\epsilon_{start}$ | 0.95 |
| Action dimension $A$ | 5 | $\epsilon_{end}$ | 0.005 |
| Hash bit (for MLP) | 16 | $decay$ | 80 |
| | | Policy net update interval $I_p$ | 1 |
| | | Target net update interval $I_t$ | 20 |

TABLE IV: Size of MLP-Based Model and Tabular Models

| **Model** | **Size Expression** | **Configuration** | **#Param/Entries** |
|---|---|---|---|
| MLP | $SH + HA + H + A$ | $H = 100$ | 1.05K |
| Table (direct) | $2^{BS}A$ | $B = 4$ | 328K |
| | | $B = 8$ | 21.5G |
| Table (token) | $2A$(#unique states) | $B = 4$ | 37.3K |
| | | $B = 8$ | 592K |

model, the state is discrete and the state space is $2^{BS}$. The theoretical storage requirement is too large (up to 21.5G), so we tokenize the unique states and use the tokens instead of the state vectors to implement table-based models.

*2) Simulator:* We evaluate our approach using ChampSim [50]. The parameter is shown in Table V. We simulate all prefetchers at the last-level cache (LLC). The replacement policy is LRU (least recently used).

TABLE V: Simulation Parameters

| **Parameter** | **Value** |
|---|---|
| CPU | 4 GHz, 4 cores, 4-wide OoO,<br>256-entry ROB, 64-entry LSQ |
| L1 I-cache | 64 KB, 8-way, 8-entry MSHR, 4-cycle |
| L1 D-cache | 64 KB, 12-way, 16-entry MSHR, 5-cycle |
| L2 Cache | 1 MB, 8-way, 32-entry MSHR, 10-cycle |
| LL Cache | 8 MB, 16-way, 64-entry MSHR, 20-cycle |
| DRAM | $t_{RP} = t_{RCD} = t_{CAS} = 12.5$ ns<br>2 channels, 8 ranks, 8 banks<br>32K rows, 8GB/s bandwidth per core |


Fig. 8: Prefetch accuracy of ReSemble and state-of-the-art prefetchers.

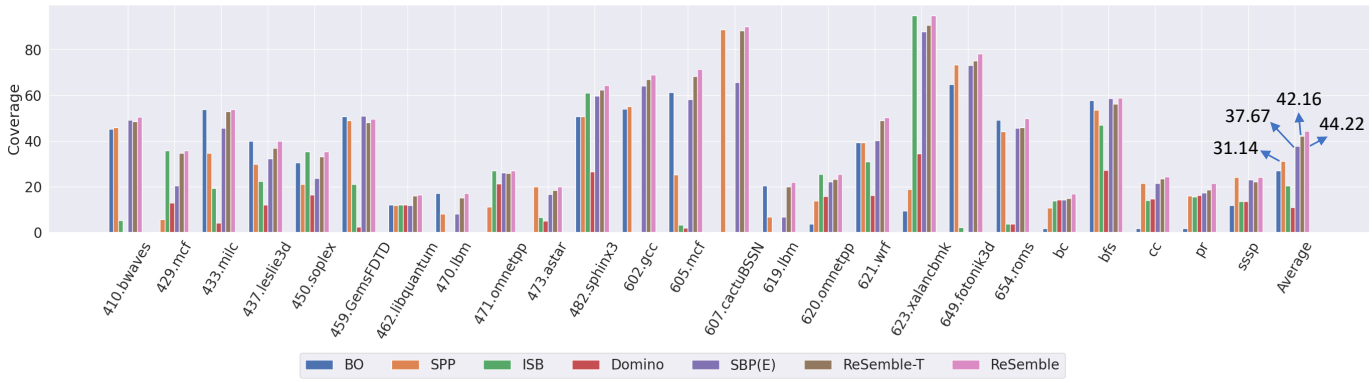



Fig. 9: Prefetch coverage of ReSemble and state-of-the-art prefetchers.

*3) Benchmarks:* We evaluate ReSemble, its tabular variant, and the baselines with benchmarks *SPEC CPU 2006* [37], *SPEC CPU 2017* [38], and *GAP* [51]. For all the applications, we simulate for 100M instructions, in which 20M instructions are the warm-up and the next 80M are for measurement.

### B. Model Learning Performance

We evaluate the model learning performance by studying the average reward for each 1K access windows. Besides the MLP-based ReSemble and its tabular variants, we also explore the influence of incorporating program counters (PC). The models we evaluated for learning performance include:

- Table-based model with 4-bit hashing and 8-bit hashing for preprocessing, with and without PC as input.
- MLP-based model with and without PC as input.

*1) Average Rewards:* The evaluation results are shown in Table VI. We have three observations from the results. First, MLP-based model without incorporating PC achieves the highest average reward for all the benchmarks. Second, while smaller hash bit compresses the state space in a higher rate, it causes notable performance drop. Third, the incorporation of program counters does provide significant contribution to the average rewards. Particularly, for SPEC06, PC impairs the performance of MLP-based model.

TABLE VI: Average Rewards of 1K Accesses Windows

| Model | | Benchmarks | | |
|---|---|---|---|---|
| **Configuration** | **PC** | ***SPEC 06*** | ***SPEC 17*** | ***GAP*** |
| Table: 4-bit hash | × | 437.97 | 440.42 | 19.93 |
| Table: 8-bit hash | × | 430.49 | 457.08 | 28.21 |
| MLP | × | **459.99** | **589.19** | **58.72** |
| Table: 4-bit hash | ✓ | 404.88 | 452.68 | 19.72 |
| Table: 8-bit hash | ✓ | 492.30 | 451.42 | 21.16 |
| MLP | ✓ | 348.35 | 535.60 | 55.29 |

*2) Case Study:* To better understand the online learning process of ReSemble, we study the rewards and actions using several cases: the same applications as in Fig 1. Fig 6 shows the learning process of various models we implemented at the first 400K accesses in the testing trace. The curves are smoothed by a factor of 10. We can observe that MLP-based model achieves notable higher rewards for *471.omnetpp* and *623.xalancbmk*. For *433.lbm*, MLP-based model shows more stable performance than tabular models, 8-bit tabular model shows higher rewards than the 4-bit tabular model. For *621.wrf*, MLP-based model without PC shows the highest rewards, while introduction of PC harms the performance of both MLP-based model and tabular model with 4-bit hash.

We show the actions of the best-performing MLP-based model and tabular model to understand the performance differ-

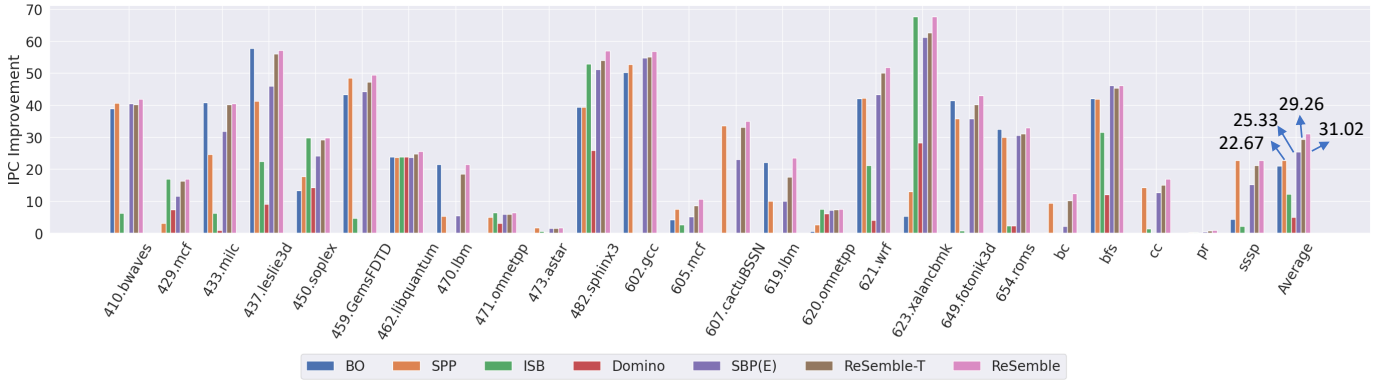



Fig. 10: IPC improvement using ReSemble and state-of-the-art prefetchers.

ence in Figure 7. First, MLP gives more frequent prefetching switches among windows observed from the first 200K accesses of *433.lbm* and *621.wrf*. This is a result of quicker response of the MLP-based model. Second, in application *471.omnetpp* and *623.xalancbmk*, MLP-based model selects the optimal prefetcher of a higher rate in a 1K window, while tabular model usually involves the selection of sub-optimal prefetchers. This is the result of better adaptability of the MLP-based model. Overall, the quicker response and better adaptability lead to higher performance of MLP-based model.

### C. Ensemble Prefetch Performance

*1) Baseline Prefetchers:* We evaluate ReSemble and the tabular variant with 8-bit hashing (ReSemble-T) using baselines including:

- **Individual prefetchers** for ReSemble input: BO, SPP, ISB, and Domino, as described in Section V-A1.
- **Sandbox Prefetcher (SBP)** [26], a state-of-the-art non-RL ensemble prefetcher. SBP uses a Bloom filter to evaluate the accuracy of multiple offset prefetchers at run-time, which can be considered as a generalization of tournament branch predictor [52]. The greedy strategy leads to the limitation of response lag (a sub-optimal prefetcher works for a period until the average performance of another prefetcher surpasses it).

SBP(E) is our implementation as an extended version of SBP. First, we use a regular buffer to store the history prefetches instead of a Bloom filter, which provides more accurate filter matching. Second, we extend the constraint of offset prefetchers to all types of prefetchers. Therefore, SBP(E) is an adaptive prefetcher that selects the prefetcher with the highest recent prefetching accuracy. We set the buffer size as 256, which is the same as a training batch in the example ReSemble for evaluation.

*2) Metrics:* To evaluate the performance of ReSemble and the baseline prefetchers, we use the following metrics:

- **Prefetch accuracy** as the ratio of useful prefetches to the overall prefetches;
- **Prefetch coverage** as the ratio of useful prefetches to the overall cache misses;
- **IPC improvement** as the percentage increase of instructions per cycle.

A useful prefetch is defined as the prefetched line being referenced by the application before it is replaced.

*3) Results:* Figure 8 illustrates the comparison of prefetch accuracy between ReSemble and state-of-the-art prefetchers. On the average, ReSemble achieves the highest accuracy at 85.27%. ReSemble-T achieves 83.94%, higher than the baselines. SBP(E), as an ensemble prefetcher, achieves 82.05%, notably higher than individual prefetchers: BO at 60.51%, SPP at 77.9%, ISB at 71.07%, and Domino at 43.25%.

Figure 9 shows the comparison of prefetch coverage between ReSemble and state-of-the-art prefetchers. ReSemble improves the coverage from 31.14% (SPP) to 41.02%, higher than SBP(E) at 37.67%. ReSemble-T achieves 42.16% coverage, slightly lower than ReSemble. The coverages of other baseline individual prefetchers are: BO at 27.04%, ISB at 20.36%, and Domino at 10.83%.

Figure 10 shows the IPC improvement contributed by prefetching. ReSemble achieves the highest IPC improvement at 29.52%, ReSemble-T achieves 29.26%, both are significantly higher than the baselines. Specifically, BO achieves 20.93%, SPP achieves 22.67%. ISB achieves 12.36%, Domino achieves 4.91%, and the ensemble baseline SBP(E) achieves 25.33%. Therefore, ReSemble outperforms the best individual prefetcher SPP by 8.35% and outperforms the ensemble baseline SBP(E) by 5.69%. ReSemble-T outperforms the best individual prefetcher SPP by 6.59%, SBP(E) by 3.93%.

## VI. Discussion

### A. Overhead analysis

*1) Latency:* A hardware implementation will incur some latency. In ReSemble, the policy net is trained online without stalling the target net (Section IV-E). The role switch design of the two MLPs also avoids the stalling of model update (Figure 4). Thus, only the forward-path inference latency is critical to prefetching. The MLP network is highly parallelizable. The end-to-end inference under complete parallelization can be estimated as:

$$T = \underbrace{T_h + T_n}_{\text{Preprocessing}} + \underbrace{\underbrace{T_{mm_h} + T_{av}}_{\text{Hidden layer}} + \underbrace{T_{mm_o} + T_{av}}_{\text{Output}}}_{\text{MLP}} + \underbrace{T_{qv}}_{\text{Action}} \quad (14)$$

where $T_h$ is the latency of hash function, $T_n$ is the normalization latency, $T_{mm_h}$ and $T_{mm_o}$ are the matrix multiplication latency of the MLP hidden and output layer, $T_{av}$ is the latency of activation functions, $T_{qv}$ is the latency of action selection that finds the maximum Q value in an MLP output vector.

TABLE VII: Estimation of ReSemble Inference Latency

| Phase | Description | Cycle |
|---|---|---|
| Prepossessing | • $T_h = \lceil \log_2 \lceil \frac{\text{Addr bit}}{\text{Hash bit}} \rceil \rceil$<br>• $T_n$: constant multiplication | 2<br>1 |
| MLP | • $T_{mm_h} = \lceil 1 + \log_2 S \rceil$<br>• $T_{mm_o} = \lceil 1 + \log_2 H \rceil$<br>• $T_{av} \times 2$: look up table | 5<br>9<br>2 |
| Action | • $T_{qv} = \lceil \log_2 A \rceil$ | 3 |
| **Total** | | 22 |

The inference latency depends on the optimization of hardware implementation. Under ideal parallel implementation, the estimated latency $T \approx 22$ cycles according to Equation 14 and Table III. The estimated latency for each component is shown in Table VII. Recent works have explored more efficient implementations of matrix multiplications by using lookup tables [53] and combinational logic [54].

To evaluate the influence of prefetch latency to the performance of prefetching, we introduce latency $T$ varied from 0 to 40 cycles in simulations. The performance of prefetching is shown in Figure 11. Considering the pipeline design in hardware implementation, we evaluate the throughput (TP) in two cases: $1/T$ (low TP) and 1 (high TP) inference per cycle.

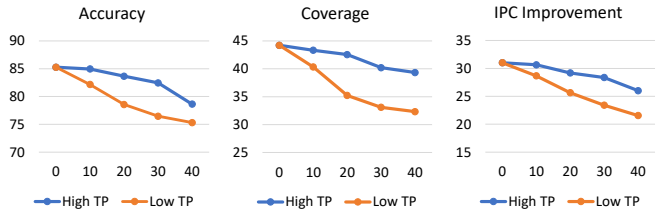



Fig. 11: ReSemble performance with prefetch latency introduced in simulation. Implementations with high throughput (High TP) and low throughput (Low TP) are evaluated.

Results show that the high throughput implementation has notably higher performance than low throughput implementation. For latency of 20 cycles, ReSemble with high TP achieves 83.66% accuracy, 42.56% coverage, and 29.19% IPC improvement, which are 4.2% higher compared with SBP in Figure 8 – 10. With higher latency, the performance of ReSemble drops. When the latency is 40 cycles, ReSemble with high TP achieves 78.64% accuracy, 39.34% coverage, and 26.02% IPC improvement, which slightly outperforms SBP which has 25.33% IPC improvement. In contrast, the performance of the low TP implementation drops quickly and shows lower IPC improvement than SBP when latency is larger than 20 cycles. Overall, for a target under parallel implementation and high-throughput pipeline with latency less than 40 cycles, we can expect ReSemble to have higher performance compared with SBP.

*2) Storage:* Table VIII shows the storage overhead of ReSemble using the configuration in Table III. There are two MLP networks requiring 4.2KB for parameters. The replay memory requires 34.8KB with a 256 entry buffer consisting of recorded prefetch addresses. Replay memory is used for training and does not require fast access. Hence, it is stored in the main memory just like prior work STMS [12] and Domino [14].

TABLE VIII: Estimation of ReSemble Storage Overhead

| Structure | Description | Size |
|---|---|---|
| MLP | • # MLP = 2<br>• # parameters = 1.05K<br>• 16-bit fixed point<br>• Stored on chip | 4.2KB |
| Replay Memory | • # entries of transitions = 2K<br>• Each entry: $(s_t, a_t, r_t, s_{t+1})$<br>• $2 \times s_t(4 \times 16\text{b}) + a_t(3\text{b}) + r_t(1\text{b})$<br>• Prefetch window: $256 \times p_t(58\text{b})$<br>• Stored off chip in main memory | 34.8KB |

## B. Incorporating NN-Based Prefetcher

Our ensemble framework is compatible with neural network (NN) based prefetchers that have been increasingly studied recently [15], [17], [33], [55]. We incorporate Voyager [33], an NN-based prefetcher using LSTM (Long Short-Term Memory) to predict memory accesses. Voyager learns from temporal patterns and predicts without the constraint of spatial range. Therefore, we replace Domino in previous experiments with Voyager here.

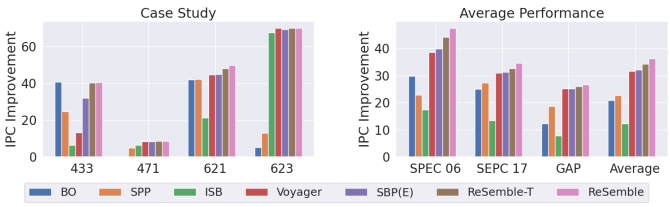



Fig. 12: Performance of ReSemble with Voyager input.

Figure 12 shows various cases and the average (geometric mean) performance of ReSemble using Voyager as a input prefetcher. It shows that Voyager cannot outperform all the prefetchers (the *433.milc*) though it is powerful. We also observe that ReSemble benefits from the power of Voyager and also makes use of other rule-based prefetchers, which leads to 36.22% IPC improvement, 4.71% higher than individual Voyager prefetcher, and 5.10% higher than the ReSemble without Voyager in Section V.

## VII. Related Work

We have compared ReSemble with state-of-the-art prefetchers [6], [8], [14], [43], an ensemble prefetcher SBP [26] in Section V, and a state-of-the-art NN-based prefetcher in Section VI-B.

Some prior works have explored the application of machine learning algorithms to data prefetching techniques. One way of using ML in prefetching is to directly learn and predict the memory access addresses, using sequence models such as LSTM [15], [16], [18], [55], [56] or attention mechanism [33], [57], [58]. These prefetchers can serve as the input of ReSemble, as discussed in Section VI-B.

Another way is to use ML algorithms to help existing prefetchers, such as providing extra prefetching reference [17] or deciding the configuration of a prefetcher or multiple prefetchers [24], [25], [59]. In contrast to those techniques, ReSemble trains an ensemble controller online without the necessity of offline training. ReSemble model decision is based on the current observation of all the prefetchers instead of the history of one prefetcher, leading to more accurate prediction and faster response.

## VIII. Conclusion

We presented ReSemble, a general ensemble framework that dynamically leverages predictions from multiple prefetchers and updates online using reinforcement learning. The keys to our approach are using *hash and norm* to reduce input memory address space, using *lazy sampling* method to handle the lag of cache feedback, and using simple MLP as the ensemble controller, which outperforms table-based controller with smaller model size. ReSemble shows 85.27% in accuracy and 44.22% in coverage. It outperforms individual prefetchers and the baseline state-of-the-art ensemble prefetcher. ReSemble contributes 31.02% improvement to IPC, which outperforms the best individual prefetcher by 8.35%, and outperforms the ensemble baseline SBP by 5.69%. In future work, we plan to explore the optimization of ReSemble hardware implementation, its sensitivity to varying budgets, and ensemble prefetching for multi-core architectures.

## Acknowledgment

We thank the anonymous reviewers for their constructive feedback. This work has been supported by the U.S. National Science Foundation under grant numbers CCF-1912680 and PPoSS-2119816.

## References

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

[2] C. Carvalho, "The gap between processor and memory speeds," in *Proc. of IEEE International Conference on Control and Automation*, 2002.

[3] S. P. Vander Wiel and D. J. Lilja, "When caches aren't enough: Data prefetching techniques," *Computer*, vol. 30, no. 7, pp. 23–30, 1997.

[4] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 252–263.

[5] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 96–96.

[6] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.

[7] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 141–152.

[8] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 247–259.

[9] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 69–80, 2009.

[10] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 252–263, 2006.

[11] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 99–108.

[12] ——, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 79–90.

[13] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 996–1008.

[14] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 131–142.

[15] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: the road to compact ml-driven prefetcher," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 461–470.

[16] A. Srivastava, T.-Y. Wang, P. Zhang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "Memmap: Compact and generalizable meta-lstm models for memory access prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2020, pp. 57–68.

[17] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "Raop: Recurrent neural network augmented offset prefetcher," in *The International Symposium on Memory Systems*, 2020, pp. 352–362.

[18] P. Zhang, A. Srivastava, T.-Y. Wang, C. A. De Rose, R. Kannan, and V. K. Prasanna, "C-memmap: clustering-driven compact, adaptable, and generalizable meta-lstm models for memory access prediction," *International Journal of Data Science and Analytics*, pp. 1–14, 2021.

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

[20] H. Devarajan, A. Kougkas, and X.-H. Sun, "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 62–72.

[21] Y. Chen, H. Zhu, H. Jin, and X.-H. Sun, "Algorithm-level feedback-controlled adaptive data prefetcher: Accelerating data access for high-performance processors," *Parallel Computing*, vol. 38, no. 10-11, pp. 533–551, 2012.

[22] M. Snir and J. Yu, "On the theory of spatial and temporal locality," Tech. Rep., 2005.

[23] C. D. Gracia *et al.*, "Ensemble prefetching through classification using support vector machine," in *Intelligent Systems Technologies and Applications*. Springer, 2016, pp. 261–273.

[24] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem, "Maximizing hardware prefetch effectiveness with machine learning," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, Aug. 2015, pp. 383–389.

[25] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 83–95.

[26] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 626–637.

[27] M. F. Uluat and V. İşler, "Ensemble adaptive tile prefetching using fuzzy logic," *International Journal of Geographical Information Science*, vol. 30, no. 6, pp. 1117–1136, 2016.

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

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

[30] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

[31] M. Maas, "A taxonomy of ml for systems problems," *IEEE Micro*, vol. 40, no. 5, pp. 8–16, 2020.

[32] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[33] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 861–873.

[34] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[36] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[37] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[38] S. CPU2017", "The standard performance evaluation corporation," https://www.spec.org/cpu2017/, 2017.

[39] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE, 1998, pp. 357–368.

[40] S. Byna, Y. Chen, and X.-H. Sun, "Taxonomy of data prefetching for multicore processors," *Journal of Computer Science and Technology*, vol. 24, no. 3, pp. 405–417, 2009.

[41] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–35, 2016.

[42] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

[43] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[44] J. Hu, H. Niu, J. Carrasco, B. Lennox, and F. Arvin, "Voronoi-based multi-robot autonomous exploration in unknown environments via deep reinforcement learning," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 12, pp. 14 413–14 423, 2020.

[45] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[46] C. Zhang, S. R. Kuppannagari, and V. K. Prasanna, "Maximum entropy model rollouts: Fast model based policy optimization without compounding errors," *arXiv preprint arXiv:2006.04802*, 2020.

[47] M. Roderick, J. MacGlashan, and S. Tellex, "Implementing the deep q-network," *arXiv preprint arXiv:1711.07478*, 2017.

[48] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," *The Third Data Prefetching Championship*, 2019.

[49] P. Zhang, R. Kannan, A. Nori, and V. Prasanna, "A2p: Attention-based memory access prediction for graph analytics," 01 2022, pp. 135–145.

[50] "ChampSim", "https://github.com/champsim/champsim," 2017.

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

[52] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.

[53] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1775–1780.

[54] M. Nazemi, A. Fayyazi, A. Esmaili, A. Khare, S. N. Shahsavani, and M. Pedram, "Nullanet tiny: Ultra-low-latency dnn inference through fixed-function combinational logic," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 266–267.

[55] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.

[56] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[57] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[58] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 103–112.

[59] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–10.

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

We evaluate our method using a modified ChampSim for ML Prefetching Competition (https://github.com/Quangmire/ChampSim). First, we use the cache miss trace dataset provided in the artifact, including SPEC 2006, SPEC 2017, and GAP. Then using the modified ChampSim with four prefetchers implemented (https://github.com/resemble1/ChampSim), we generate the prefetching suggestions by all the prefetchers.

Using the above 1) cache miss trace and 2) generated prefetching suggestions as input, we simulate the online learning process with the proposed approach ReSemble (https://doi.org/10.5281/zenodo.6462850). ReSemble will generate the final prefetching address based on the prefetching suggestions from all input prefetchers and the interactions with cache miss trace.

Using the final prefetching address, we simulate the prefetching performance with the modified ChampSim (https://github.com/resemble1/ChampSim) and generate reports including accuracy, recall, and IPC improvement.

We use a rule-based resemble method SBP as the baseline. There is no open-source artifact for this method so we implemented our version at: https://github.com/resemble1/SBP-E.git.

The cache miss traces are large, and the generation of prefetching files is time-consuming, so we provided a piece of sample data in the ReSemble Container (https://hub.docker.com/r/resemble1/resemble) with all dependencies installed.

The platform information we use:

(1) CPU: Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz.
(2) NVIDIA GP102 TITAN Xp.
(3) Operating systems and versions: Ubuntu 20.04.1 LTS.
(4) Compilers and versions: g++ 9.4.0; python 3.8.13
(5) Libraries and versions: pandas 1.4.1; numpy=1.21.5; pytorch=1.10.2; tqdm=4.63.0

## AUTHOR-CREATED OR MODIFIED ARTIFACTS:

**Artifact 1**
Persistent ID: `https://doi.org/10.5281/zenodo.6462850`
Artifact name: ReSemble Source Code

**Artifact 2**
Persistent ID: `https://hub.docker.com/r/resemble1/resemble`
Artifact name: ReSemble Container

**Artifact 3**
Persistent ID: `https://github.com/resemble1/ChampSim`
Artifact name: Modified ChampSim for ReSemble
Citation of artifact: https://github.com/Quangmire/ChampSim

**Artifact 4**
Persistent ID: `https://github.com/resemble1/SBP-E.git`
Artifact name: SBP-E

*Reproduction of the artifact with container:* Following the steps below to run the ReSemble demo:

1) Download the container: docker pull resemble1/resemble:0415

2) Start the container: a) Run the contariner: docker run -itd resemble1/resemble:0415 b) Check the container ID: docker ps c) Execute the container: docker exec -it [CONTAINER ID] /bin/bash

3) Activate environment in container: conda activate sc22

4) Go to the working directory: cd /root/sc22/ReSemble/

5) We provide a generated sample prefetching data for running the demo based on a short trace from 654.roms in SPEC 2017, stored in /root/sc22/ReSemble/sample_data/

6) Run MLP-based ReSemble demo: cd /root/sc22/ReSemble/ReSemble_MLP; ./run_demo.sh

7) Run tabular variant of ReSemble demo: cd /root/sc22/ReSemble/ReSemble_Tab; ./run_demo.sh

8) Results of demos are stored in /root/sc22/ReSemble/results/, where .pkl file is the models, .pref.txt file is the prefetch addresses for simulation, .rewards.csv is the rewards and proportions of actions for the demo prefetching process.

9) Check the ReSemble online learning records by printing the .rewards.csv files on the screen. Specifically, "654.roms-s0.mlp.rewards.csv" shows the MLP-based model results; "654.roms-s0.tab-4.rewards.csv" shows the tabular variant using 4-bit hashing.