Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies

Anastasios Zouzias
Huawei Technologies
Zurich Research Center
Switzerland
anastasios.zouzias@huawei.com

Kleovoulos Kalaitzidis
Huawei Technologies
Zurich Research Center
Switzerland
kleovoulos.kalaitzidis@huawei.com

Boris Grot
University of Edinburgh
School of Informatics
United Kingdom
boris.grot@ed.ac.uk

Abstract—**Recent years have seen stagnating improvements to branch predictor (BP) efficacy and a dearth of fresh ideas in branch predictor design, calling for fresh thinking in this area. This paper argues that looking at BP from the viewpoint of Reinforcement Learning (RL) facilitates systematic reasoning about, and exploration of, BP designs. We describe how to apply the RL formulation to branch predictors, show that existing predictors can be succinctly expressed in this formulation, and study two RL-based variants of conventional BPs.**

I. Introduction

Branch prediction (BP) lies at the heart of high-performance processor design as it enables larger instruction windows that are imperative for extracting instruction- and memory-level parallelism. Today's state-of-the-art branch predictors learn correlations between branches through the use of large hardware tables, with efficacy strongly correlated with the amount of storage available. However, with the looming end of Moore's law, processor designers are faced with the challenge of improving performance, including that of branch predictors, without the benefit of larger transistor budgets.

Today, virtually all branch predictors in high-end processors are highly-engineered variants of TAGE [12] and Perceptron [7]. Despite extensive differences in the prediction mechanisms of these two families, predictors in each of these families are produced through meticulous feature engineering, often driven by a combination of intuition and simulator-driven "tweaking". Perhaps not surprisingly, improvements to both families of BPs have stagnated in recent years, and the question facing the entire processor industry is how to uncover new optimizations and entirely new designs that can outperform today's best.

This paper observes that branch prediction can be expressed as a *Reinforcement Learning (RL)* problem. In RL, the learning agent seeks to find a (near-)optimal policy that maximises the reward function by interacting, and learning from, the environment [15]. Viewed through this lense, a branch predictor is an agent that observes the program's control flow (i.e., the history of branch outcomes) and tries to learn a policy that maximizes the accuracy of future control-flow predictions. We argue that viewing BP as an RL problem enables a systematic approach to model and explore branch predictor designs through explicit reasoning of each aspect of the predictor, such as its state representation, decision-making policy, and strategy to minimize the number of mispredictions. We showcase how existing predictors seamlessly map to the RL formulation, and present two RL-based variants of predictor designs. Finally, we highlight new promising research venues that we believe RL opens for BP.

II. Reinforcement Learning Background

Reinforcement learning algorithms are well-suited for scenarios where an *agent* can learn from an *environment*. The agent interacts with the environment using a set of actions. After the agent selects an action, the environment responds with the action's reward and the next environment state. The state set and state transitions of the environment are commonly modelled with a *Markov Decision Process*[1] (MDP) [15]. The goal of an agent is to take actions that maximize the future cumulative reward G_t for any time-step t, i.e., $G_t := R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

RL agents are objective-driven and characterized by their *policy* (behavior) on any given state. A policy π is a probability distribution over actions given a state, i.e., $\pi(a|s) := \mathbb{P}\left(A_t = a | S_t = s\right)$. The state-action value function $Q_\pi(s,a)$ (known as Q-values) is the expected G_t starting from s, taking an action a, and then following the policy π, i.e.,

$$Q_\pi(s,a) := \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right]. \quad (1)$$

Given the state-action value function, the *greedy* policy is defined as $\pi_{\text{greedy}}(s) := \arg\max_{a\in\mathcal{A}} Q(s,a)$. Similarly, for $0 \leq \varepsilon < 1$, the *ε-greedy* policy is a policy that equals $\pi_{\text{greedy}}(s)$ with probability $1-\varepsilon$ and selects an action uniformly at random with probability ε.

Depending on whether the agent models the environment (i.e., models the reward function and/or transition probabilities), RL methods are categorized in *model-based* and *model-free*. In this work, we focus[2] on the model-free category in

[1]A *Markov Decision Process* is a tuple ($\mathcal{S}$,$\mathcal{A}$,P,$\mathcal{R}$, γ), where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a finite set of actions, P is the state transition probability matrix $P^a_{ss'} = \mathbb{P}\left(S_{t+1} = s' | S_t = s, A_t = a\right)$, $\mathcal{R}$ is a reward function $\mathcal{R} := \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a\right]$ and $\gamma \in [0,1)$ is a discount factor where $\mathbb{E}\left[\cdot|\cdot\right]$ denotes the conditional expectation.

[2]Model-free aim to *only* learn an optimal policy, i.e., similarly to branch prediction that aims for a specific speculation policy, whereas model-based methods aim to also model the environment.

which there are two main types of RL methods: *tabular* (Section II-A) and *function approximation* (Section II-B). In tabular methods, the agent tries to learn from the environment by storing/updating information in fixed-sized tables where each entry corresponds to a state/action pair. In function approximation methods, the agent directly optimizes a parametrized function of the policy.

A. Tabular Methods: Q-Learning

A number of tabular RL methods exist; most popular ones include TD-learning [15], SARSA [14], Q-Learning [17] and double Q-Learning [6]. Here we focus on the Q-Learning algorithm that provides specific convergence guarantees [17][3].

Q-Learning stores the Q-values $Q(s,a)$ for every state and action pair in a fixed-sized table. Given a state s from the environment, Q-Learning predicts the action greedily using the policy $\pi_{\text{greedy}}(s)$. The Q-Learning update rule works as follows: (a) choose an $a \in \mathcal{A}$ based on the current state using the ε-greedy policy; (b) observe reward r, next state s' after action a; (c) update $Q(s,a) \mathrel{+}= \alpha[r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)]$ for a positive learning rate parameter α.

B. Function Approximation Methods: Policy Gradient

In function approximation methods, the policy is parametrized using a vector parameter $\boldsymbol{\theta}$ and is denoted by $\pi_{\boldsymbol{\theta}}(a|s)$. The goal is to optimize the expected cumulative reward by maximizing

$$J(\boldsymbol{\theta}) := \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi_{\boldsymbol{\theta}}}(s,a), \tag{2}$$

where $d^{\pi}(s)$ is the stationary distribution[4] of the Markov Chain defined by $\pi_{\boldsymbol{\theta}}$ on a given MDP. Using gradient ascent, $\boldsymbol{\theta}$ can be updated towards the direction of the gradient $\nabla J(\boldsymbol{\theta})$ to find an (local) optimal $\boldsymbol{\theta}$ that maximizes the average future cumulative reward, i.e., $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla J(\boldsymbol{\theta})$.

As a representative example, the REINFORCE Policy-Gradient method [15], [18] works as follows:

- Collect reward r on state $s \in \mathcal{S}$ with action $a \in \mathcal{A}$.
- Update policy by maximizing $J(\boldsymbol{\theta})$ via an optimization strategy (i.e., online gradient ascent).

Given a discrete action space (for example of size two), the policy is usually parametrized by a softmax function $\pi_{\boldsymbol{\theta}}(a|s) := \mathrm{e}^{h(s,a,\boldsymbol{\theta})} / \sum_{a\in\mathcal{A}} \mathrm{e}^{h(s,a,\boldsymbol{\theta})}$ for an arbitrary differentiable function $h(s,a,\boldsymbol{\theta})$.

The second step of the Policy-Gradient method requires a gradient computation. In Section IV-B, a special case will be considered where h is a linear function on $\boldsymbol{\theta}$, i.e., $h(s,a,\boldsymbol{\theta}) := \boldsymbol{\theta}^{\top}\boldsymbol{x}(s,a)$, where $\boldsymbol{x}(s,a)$ is a vector representation of the state-action space and $\boldsymbol{\theta}^{\top}\boldsymbol{x}$ denotes vector dot-product. In this specific case, the policy gradient theorem [15, Chapter 13] gives us a closed form solution for the gradient, i.e., $\nabla J(\boldsymbol{\theta}) = \boldsymbol{x}(s,a) - \boldsymbol{\mu}$, where $\boldsymbol{\mu} := \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s)\boldsymbol{x}(s,a)$.

[3]We have experimented with most of the tabular methods and found their branch prediction performance to be similar.

[4]In general, a branch predictor might never converge to a stationary distribution due to the partial-observability of the state space. As an example, consider data-dependent branches whereby branch outcomes are correlated with data values but the actual data values are not visible to the branch predictor.

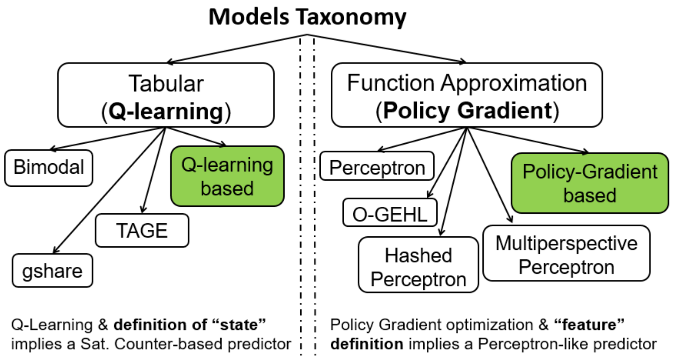
Models Taxonomy
Tabular (Q-learning)
Function Approximation (Policy Gradient)
Bimodal
Q-learning based
TAGE
gshare
Perceptron
O-GEHL
Hashed Perceptron
Multiperspective Perceptron
Policy-Gradient based
Q-Learning & definition of "state" implies a Sat. Counter-based predictor
Policy Gradient optimization & "feature" definition implies a Perceptron-like predictor

Fig. 1. Taxonomy of model-free RL methods and branch predictors.

III. Reinforcement Learning for Branch Prediction

A. Branch Prediction as an RL Problem

In this section, we show that branch prediction is a task that can be seamlessly formulated as an RL problem as it abides by similar theoretical principles. In particular, through the lenses of RL, branch prediction is viewed as a pure online optimization task where the high-level goal is to minimize the rate of branch mispredictions. In RL parlance: *"Maximization of the future cumulative reward equals to minimization of future cumulative branch mispredictions."*

The branch predictor corresponds to the *agent* and the processor's execution environment (e.g., program counter, registers, memory/cache content, etc) corresponds to the *environment*. The *environment* is private and communicates its state to the branch-predictor agent so that the latter can decide which action a from the action space $\mathcal{A} := \{\mathrm{T}, \mathrm{NT}\}$[5] will be followed. Depending on the correctness of the chosen action, the environment responds back with a reward $R \in \mathbb{R}$ that guides the predictor's update (e.g., confidence counter increase or reset on a, respectively, correct or incorrect prediction). At that point, the next state of the environment is also disclosed. Generally, the state content can consist of any information available to the BP; e.g., the branch PC address, the local and/or global branch history, etc. For instance, if the global branch history is part of the state definition, its instance after a prediction (next state) will include the predicted branch outcome (note that this is the common case of speculative history update in BP).

In analogy with RL methods, branch predictors can be categorized into two separate classes: tabular and function-approximation models. Tabular predictors model their actual predictions with a set of finite-state machine (FSMs) represented by up/down saturating counters stored in storage-constrained tables. Their goal is to couple branches with the correct FSM by matching patterns in the execution history. Examples of such predictors include the Bimodal predictor [13],

[5]T stands for *taken*, NT for *not-taken*.

two-level branch predictors, such as gshare [10], and, most recently, the TAGE-based [12] predictors.

Meanwhile, existing branch predictors that fall under the function-approximation models are mainly variants of the Perceptron [7] predictor, which is based on the homonymous online learning algorithm [1]. Perceptron learns from previous experiences by training a single-layer neural network, which stands in contrast to the memoization approach of tabular models. O-GEHL [11], Hashed Perceptron [16], and, most recently, Multiperspective Perceptron [8], are examples of this class of predictors.

In Figure 1, we classify common branch predictors into the two categories of tabular and function-approximation models. We respectively map them to their RL-based counterparts of either Q-Learning or Policy-Gradient origin. In essence, tabular predictors that are based on N-bit saturating counters can be formulated with a Q-Learning based scheme, while function-approximation models are a good fit for Policy-Gradient methods. We explain this mapping for Q-Learning based predictors in the next section; Policy-Gradient based schemes are analyzed later in Section IV.

B. An Illustrative Example: Q-Learning Based BP

We now give a detailed example of how an existing tabular branch predictor can be cast in RL terms. For illustrative purposes, we focus on relatively simple predictors – Bimodal and gshare (Section IV-B discusses a more complex Perceptron predictor). We opt not to include TAGE in our exploration as its design principles originate from the method of partial string matching used in data compression [4]. However, a similar RL-based optimization of TAGE could be feasible; we leave this case for future work.

Both Bimodal and gshare can be viewed as Q-Learning instances; for Bimodal the state space is the set of branch PC addresses, while for gshare it is the set of branch PC addresses exclusive-ORed with (a part of) the global branch-history register (GHR). In this simple form, the RL-based versions of these predictors do not perform any exploration (see section IV-D), that is, $\varepsilon = 0$. According to the description of section II-A, their update rule will be reduced to $Q(s,a) = (1-\alpha)Q(s,a) + \alpha r$, since future rewards are not considered, and thus $\gamma = 0$.

We have implemented in the CBP-5 [3] framework the aforementioned Q-Learning based variant of gshare, which we call *G-QLAg* (Global-History Q-Learning Agent). G-QLAg consists of a N-entry table, each containing two Q-values: Q_{T} and Q_{NT}. Q-values are signed floating-point numbers ranging from -1 to 1. Experimentally, we found that using 6-bit Q-values is a good trade-off in our setup. Like gshare, G-QLAg is indexed with a hash of the branch PC and GHR modulo the size of the prediction table. For the matching entry, when $Q_{\mathrm{T}} > Q_{\mathrm{NT}}$, the prediction is *T*. Similarly, when $Q_{\mathrm{T}} < Q_{\mathrm{NT}}$, the prediction is *NT*. In the case where $Q_{\mathrm{T}} = Q_{\mathrm{NT}}$, the predicted direction (*T, NT*) is picked at random with equal probability. Q-values are initialized to 0. At update time, the reward r is set to 1 or -1 if the prediction was correct or incorrect,

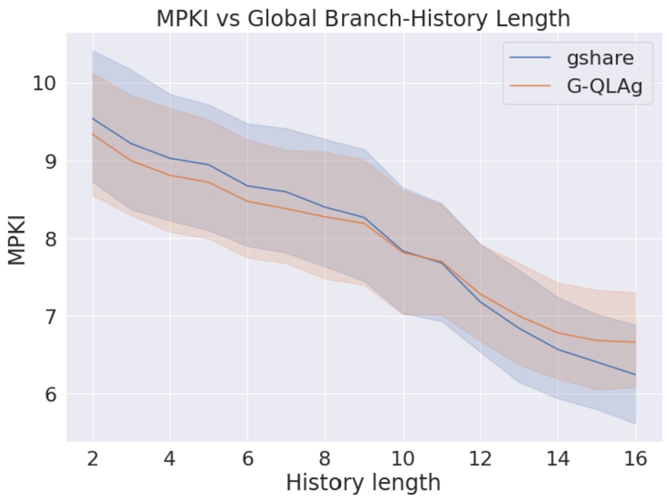

Fig. 2. MPKI of gshare vs G-QLAg over all CBP-5 traces [3] as a function of global branch-history length. The error bands are proportional to the standard deviation over all traces.

respectively. The Q-value that corresponds to the prediction $(\max(Q_{\mathrm{T}}, Q_{\mathrm{NT}}))$ is updated according to the formula in the previous paragraph with learning rate $\alpha = 0.2$.

Figure 2 compares the mispredictions per kilo-instructions (MPKI) of gshare and G-QLAg by varying the global branch-history length. We consider the same storage budget (64KB) for both predictors and configure their number of entries accordingly. Our implementation of G-QLAg uses 12 bits (2 Q-values) per entry, resulting in significantly fewer entries than gshare, which requires only 2 bits per entry.

As the figure shows, the two predictors perform similarly across the range of histories. As expected, for both predictors, MPKI reduces as history length is increased. However, at longer histories, the capacity disadvantage of G-QLAg becomes more pronounced, resulting in higher MPKI than gshare.

As the study shows, it is indeed straightforward to cast existing branch predictors as RL, and can be done without considering RL aspects such as γ and $\varepsilon = 0$ for G-QLAg. However, as we show in section IV-A, considering these and other aspects of RL opens new venues for BP policy optimization.

IV. RL-based Branch Predictors: looking ahead

A. Exploring the BP Design Space through RL Methods

Following the Policy-Gradient framework explained in Section II-B, we next discuss how RL can facilitate the development of future BPs. We identify four key ingredients that a branch predictor needs to specify: (1) policy, (2) state representation, (3) loss function optimized by the predictor and (4) optimization strategy.

Policy: For branch prediction, the policy $\pi(a|s)$ is a function from the current state to a branch outcome. The policy may be expressed as a linear model or a non-linear model. Linear models are simple and might be readily amenable to a hardware-friendly implementation. The drawback of linear models is

Branch Predictor	State space ($\mathcal{S}$)	Policy (Q-values / $\pi(a\|s)$)	Objective / Loss	Online Policy Optimizer
Bimodal [13]	PC	2-bit saturating counter	N/A	Approx. Q-Learning update
Gshare [10]	PC $\oplus$ GHR	2-bit saturating counter	N/A	Approx. Q-Learning update
Perceptron [7]	PC, GHR	$\mathbf{sign}(b + w_1 ghr_1 + \cdots + w_l ghr_l)$	Hinge Loss	Perceptron update
Hashed Perceptron (O-GEHL [11])	PC, GHR	As in Perceptron but hashed PC, GHR	Hinge Loss	Perceptron update
Multiperspective Perceptron [8]	Many	Various branch features (ghist, pathhist, etc)	Hinge Loss	Perceptron update & heuristics
BranchNet [19]	PC, GHR	Convolutional Neural Network (CNN)	Logistic Loss	**Offline** Adam optimizer [9]
PolGAg	PC, GHR	$\sigma(b + w_1 ghr_1 + \cdots + w_l ghr_l)$	Equation 2	REINFORCE (Section II-B)

TABLE I
BRANCH PREDICTORS VIEWED FROM AN RL PERSPECTIVE. CATEGORIZATION IS BASED ON STATE REPRESENTATION, POLICY FUNCTION, LOSS FUNCTION AND ONLINE POLICY OPTIMIZATION. $\sigma(x) := 1/(1+\exp(-x))$ IS THE SIGMOID FUNCTION.

that they are unable to capture non-linear correlations, such as an XOR relationship. The Perceptron BP is an example of a design that employs a linear model and, hence, fails to capture correlations between non-linearly separable branches [7].

Non-linear models, including multi-layer neural networks, are fundamentally more expressive than linear ones and have the capacity to capture non-linear correlations. The limitation of non-linear models, however, is that they do not land themselves to a straight-forward microarchitectural implementation. For instance, the recent BranchNet predictor [19] attempts to overcome implementation limitations with a specialized software-hardware co-design: a plurality of convolutional neural networks (CNNs) are trained at compile-time (for up to 18 hours depending on the application) for a few hard-to-predict (H2P) branches. BranchNet's inference engine is implemented in hardware for predicting only those branches, alongside TAGE. While the design is shown to be effective, it is complex. We believe that finding the right balance between model simplicity and its capacity will be a major focus in future ML-inspired microarchitectural research.

State representation: In the context of branch prediction, the state may encompass branch addresses, global and/or local history bits, loop counters, etc. In Section II, we briefly presented examples of Bimodal and gshare BPs, which can be viewed as instances of Q-Learning with PC and PC$\oplus GHR$ state representation, respectively.

Loss function: Also known as an objective function, this governs the goal that the policy seeks to achieve. For a branch predictor, this should be minimizing the number of mispredictions. For instance, the Perceptron BP minimizes the number of mispredictions explicitly using the hinge loss function[6]. However, the goal of a branch predictor does not have to *explicitly* minimize the mistakes, but it could do so *implicitly*. That is, an alternative goal is to minimize the *misprediction probability* for a branch predictor that is probabilistic, i.e., returns the probability that the branch will be taken. Such a branch predictor optimizes the probability of predicting the correct branch direction and thus implicitly minimizes mispredictions. In Section IV-B, we will discuss the design and performance of such a predictor.

Optimization strategy: In RL, the optimization strategy consists of an *optimizer*, which is a method to minimize the loss function, along with additional "knobs" that, e.g., control the learning rate. In the BP domain, the optimization strategy defines the branch predictor update rule. For example, the Perceptron BP uses the Perceptron update rule, which is implicitly an online gradient descent optimizer with hinge loss (see Table I). In the O-GEHL predictor [11], the dynamic threshold that guides the predictor's update can be roughly thought as an adaptive learning-rate adjustment for the optimizer.

Table I summarizes these design elements for a selection of well-known branch predictors, as well as PolGAg, a Policy-Gradient based branch predictor introduced in the next section.

B. Example: A Policy-Gradient Based Branch Predictor

In this section, we study the Perceptron BP through the perspective of Policy Gradient, with the aim of showcasing how the RL construction can be used to explore the BP design-space.

To that end, we design a Perceptron-like branch predictor, which we call *Policy Gradient Agent BP* (PolGAg). The Policy Gradient Agent is based on the REINFORCE algorithm [15], a well-known Policy-Gradient method. Specifically, we adjust the REINFORCE algorithm for the context of the Perceptron BP. The resulting design is described with Algorithm 1. We derive Algorithm 1 from a holistic optimization view of the problem, i.e., maximizing the objective function of Equation 2 and making specific decisions on the definition of the policy $\pi_{\boldsymbol{\theta}}(a|s)$.

Algorithm 1 Policy Gradient Agent BP (PolGAg)
```
1: procedure PolGAg($l$, $\alpha$)   ▷ $l$ GHR bits/learning rate $\alpha$.
2:     Set $\boldsymbol{\theta}[PC] = \mathbf{0}$ for every PC.
3:     for each branch PC do
4:         Let $a \in \{\mathrm{T}, \mathrm{NT}\}$ with reward $r \in \{\pm 1\}$.
5:         $\boldsymbol{\theta}[PC] \mathrel{+}= \alpha r \pi_{\boldsymbol{\theta}}(\bar{a}|s)\boldsymbol{x}(s,a)$.
6:     end for
7:     Output: Branch prediction policy: $\pi_{\boldsymbol{\theta}}(a|s)$.
8: end procedure
```

PolGAg represents a specific design point with a policy, state space, loss function and optimization strategy (Section IV-A). It suffices to define the policy and the state space $\mathcal{S}$, since the REINFORCE method already specifies the loss function (Equation 2) and the optimization strategy (online gradient ascent). For the policy, we use a linear softmax policy function

[6]Hinge loss for a binary classifier with output y and target $t \in \{\pm 1\}$ is defined as $\max(0, 1 - ty)$.

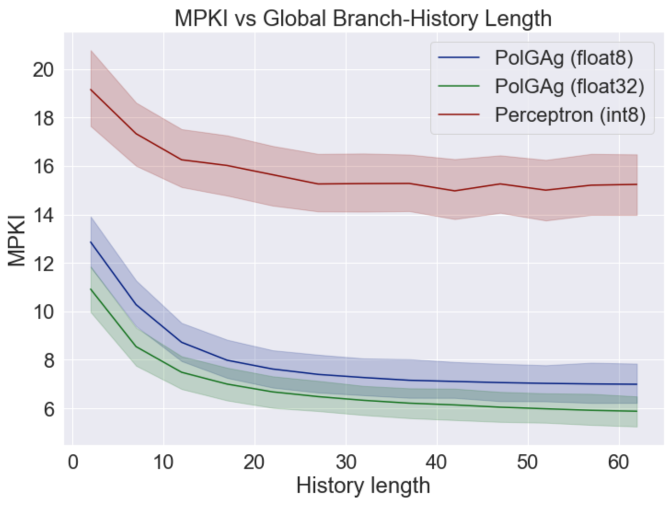

Fig. 3. MPKI for PolGAg and Perceptron predictors as a function of history length over all CBP-5 traces. Learning rate parameter α is 0.01.

with $h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \boldsymbol{x}(s,a)$. Recall that $\boldsymbol{x}(s,a)$ is a vector representation of the state-action space. The state space $\mathcal{S}$ is defined as the set of all branch addresses (PC) and the content of GHR with length l. GHR is viewed as a vector $\boldsymbol{q} \in \{\pm 1\}^l$. For the purpose of the analysis, assume that there are p unique branch addresses during the whole program execution. Also, let $\text{OHE}(PC) \in \{0,1\}^p$ be the one-hot encoding[7] (OHE) representation of the branch address. Moreover, define $\boldsymbol{q}(\text{T}) := [1, \boldsymbol{q}]$ and $\boldsymbol{q}(\text{NT}) = -\boldsymbol{q}(\text{T})$, where the constant 1 corresponds to a bias term. Finally, define $\boldsymbol{x}(s,a)$ as the $(p(l+1))$-vector $\boldsymbol{x}(s,a) = \text{OHE}(PC) \otimes \boldsymbol{q}$, where $\otimes$ denotes the Kronecker product. $\boldsymbol{x}(s,a)$ is defined for every state and action. Using the above policy and state representation, the REINFORCE method translates to a specific update step of our PolGAg Algorithm. Continuing from Section II-B, a direct calculation shows that $\nabla J(\boldsymbol{\theta}) = \boldsymbol{x}(s,a) - \boldsymbol{\mu} = 2\pi(\bar{a}|s)\boldsymbol{x}(s,a)$, where $\bar{a}$ flips the action from T to NT or vice-versa. Therefore, Step 5 of Algorithm 1 becomes $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + 2\alpha r \pi(\bar{a}|s)\boldsymbol{x}(s,a)$.

In Figure 3, we compare the MPKI of Perceptron [7] and PolGAg predictors. To avoid aliasing effects, we consider both predictors with unbounded storage, i.e., each branch has a dedicated set of weights. Firstly, for both predictors, each weight entry is stored using 8-bits. For PolGAg, we use 8-bit floats (1-bit sign, 5-bit exponent, 2-bit Mantissa) and for Perceptron 8-bit integers. As such, PolGAg has more precision by employing floats, but also a higher computation complexity. However, as this is a limit study, we ignore the higher computational requirements of PolGAg.

As the figure shows, PolGAg has a much higher prediction accuracy than the Perceptron. We attribute the difference to two factors: (a) the use of floats for PolGAg and (b) the use of logistic loss and policy gradient update versus Perceptron's hinge loss update. Unfortunately, it is not easy to quantify the effect of each of these factors separately, since the choice of the loss function and update rule is coupled. In terms of using floats vs ints, the update rule of PolGAg involves multiplication with a prediction probability, which makes it unfriendly to an integer representation. Instead, we assess a variant of PolGAg that uses float32, rather than float8, for each weight. We observe that the use of higher precision further improves PolGAg's accuracy, reducing MPKI by a full point compared to PolGAg-float8, which leads us to conclude that numerical precision does play a role in predictor's efficacy.

To summarize, we showed that the Perceptron predictor can be viewed as a Policy-Gradient method by appropriate selection of policy, state space, loss function and optimization strategy. While the resulting predictor, PolGAg, was shown to outperform the baseline Perceptron, the former may be implementation-unfriendly due to its use of floats. Rather than espousing one design or the other, our purpose is to highlight the fact that viewing branch predictors through the RL prism opens new optimization opportunities.

C. Additional Examples

We briefly show two more branch predictors that can be viewed in the Policy-Gradient framework:

O-GEHL [11]. Here, we demonstrate that the O-GEHL BP is an instance of the Policy Gradient framework. The parameter space $\boldsymbol{\theta}$ of O-GEHL is the concatenation of the 8 tables each of which have roughly $8K$ 8-bit entries. For any branch address (PC) and GHR instance, $\boldsymbol{x}(s,a)$ has only 8 non-zero entries: one per table. The policy optimizer is the Perceptron update rule, the loss function is hinge loss.

Multiperspective Perceptron [8]. This latest Perceptron variant can also be viewed in the policy optimization setting. The only difference with our previous example is the state representation (more data must be collected to be able to compute the features) and the need of a more elaborate feature engineering, i.e. definition of the function $\boldsymbol{x}(s,\alpha)$.

D. Speculation & RL Exploration

RL methods allow the agent to explore unknown states, potentially by making a decision likely to be disadvantageous in the short term, in order to achieve a higher cumulative long-term reward. We posit that such an approach could benefit existing or new BP schemes. In the RL-based BP paradigm, the exploration process would effectively require the agent (predictor) to consider information from wrong execution paths. But how to explore a wrong execution path without greatly affecting processor performance?

We observe that on a misprediction, the BP continues to make predictions (and, accordingly, advance processor's control flow) up to the point of the pipeline flush. In today's BP designs, whatever was learned by the BP on the mispredicted path is discarded. However, the path observed under a misprediction may contain information valuable in later phases of the execution. In RL terms, there exists an opportunity to improve prediction accuracy through exploration and learning under a misprediction. In the case of Q-Learning, for example, to fully apply its update rule (Sec. II-A), it would be necessary

[7]In the one-hot-encoding representation, each coordinate corresponds to a unique address. $\text{OHE}(PC)$ has 1 in the coordinate that corresponds to PC and 0, otherwise.

to consider in the predictor's update the first branch address on the wrong path. In this way, the agent would be able to perform *exploration* of the control-flow path according to the definition of the ε-greedy update step. To the best of our knowledge, the only proposed branch predictor that considers a similar option is the prophet/critic hybrid scheme [5]. We believe that the RL formulation opens the door to more systematic studies of exploration.

V. Conclusion & Future Work

This paper makes the case for *Reinforcement Learning* based branch prediction as a new way of systematically developing future BPs. Our first-stage exploration demonstrates how RL concepts map naturally to the BP principles, how existing BP designs can be cast as RL models, and how such a formulation can open untapped opportunities. Our future work will focus on further exploring the opportunities offered by the RL formulation, and on converting these opportunities into hardware-friendly implementations.

References

[1] H. D. Block, "The perceptron: A model for brain functioning. i," *Rev. Mod. Phys.*, vol. 34, pp. 123–135, Jan 1962.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[3] CBP-5, "Championship branch prediction," *In conjunction with the International Symposium on Computer Architecture https://www.jilp.org/cbp2016/*, 2016.

[4] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[5] A. Falcon, J. Stark, A. Ramirez, K. Lai, and M. Valero, "Prophet/critic hybrid branch prediction," in *International Symposium on Computer Architecture (ISCA)*, 2004, pp. 250–261.

[6] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2010, pp. 2613–2621.

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

[8] D. A. Jiménez, "Multiperspective perceptron predictor," in *Proc. 5th Championship on Branch Prediction*, 2016.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." *CoRR*, vol. abs/1412.6980, 2014.

[10] S. Mcfarling, "Combining branch predictors," Digital Equipment Corporation, Western Research Lab, Tech. Rep., 1993.

[11] A. Seznec, "Analysis of the O-GEometric history length branch predictor," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 394–405.

[12] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, 2006. [Online]. Available: http://www.jilp.org/vol8

[13] J. E. Smith, "A study of branch prediction strategies," in *International Symposium on Computer Architecture (ISCA)*, 1981, p. 135–148.

[14] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems (NIPS)*, 1996, pp. 1038–1044.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[16] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, p. 280–300, Sep. 2005.

[17] C. J. C. H. Watkins and P. Dayan, "Q-learning," in *Machine Learning*, 1992, pp. 279–292.

[18] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, p. 229–256, May 1992.

[19] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.

VI. Appendix: Exercising Policy Design in a Reinforcement learning gym

RL methods adhere to a well-defined interface with any possible environment (the agent **acts** on the environment which in turn **steps** to the next state after providing a reward on the chosen action). Based on this interface, an open-source framework called Gym [2] has been developed to facilitate research of new RL algorithmic approaches on various environments, such as bipedal robot walking (Humanoid-v2) and playing a specific video game (Pong-ram-v0) to name a few. As a major benefit, Gym allows the implementation of custom environments for new problems where RL methods could be found useful to address them. Similarly, we have extended[8] Gym by defining several custom BP environments (*gym_bp_cbp2016*) based on the public traces of CBP-5 to explore new RL policies on branch prediction.

Each of these environments is defined for a specific branch instruction of a given trace from all the 440 traces of CBP-5. For instance, the code snippet below initializes an environment for branch with PC address equal to 65872 of *SHORT_MOBILE-42* trace.

```
import pandas as pd
import gym
import gym_bp_cbp2016

# Set trace, branch and global history length
df = pd.read_parquet("SHORT_MOBILE-42.parquet")
branch = 65872
ghr_len = 17

# Initialize gym_bp_cbp2016 environment
env = gym.make('bp-cbp2016-v0')
env.init(df, ghr_len, branch)
```

An RL-based BP interacts with the *gym_bp_cbp2016* environment via the step method as listed below. The branch predictor inputs the branch direction (T/NT) as an integer type with 0/1 values and the environment responds with the next observation (GHR bits), the reward (± 1) and a boolean flag 'done' if the environment has reached the end. The RL-based BP now has access to observations and rewards and it can update its policy and make the next prediction.

```
# Predict branch direction and environment step
pred_branch_dir = agent.act(previous_obs)
obs, reward, done, _ = env.step(pred_branch_dir)
```

The above code snippet shows a branch prediction step followed by a next state step of the environment.

[8]The tool is available to download on the workshop's webpage: https://sites.google.com/view/mlarchsys/isca-2021.