Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera[1] Konstantinos Kanellopoulos[1] Anant V. Nori[2] Taha Shahroodi[3,1]
Sreenivas Subramoney[2] Onur Mutlu[1]

[1]ETH Zürich [2]Processor Architecture Research Labs, Intel Labs [3]TU Delft

ABSTRACT

Past research has proposed numerous hardware prefetching techniques, most of which rely on exploiting one specific type of program context information (e.g., program counter, cacheline address, or delta between cacheline addresses) to predict future memory accesses. These techniques either completely neglect a prefetcher's undesirable effects (e.g., memory bandwidth usage) on the overall system, or incorporate system-level feedback as an afterthought to a system-unaware prefetch algorithm. We show that prior prefetchers often lose their performance benefit over a wide range of workloads and system configurations due to their inherent inability to take multiple different types of program context and system-level feedback information into account while prefetching. In this paper, we make a case for designing a holistic prefetch algorithm that learns to prefetch using multiple different types of program context and system-level feedback information inherent to its design.

To this end, we propose *Pythia*, which formulates the prefetcher as a reinforcement learning agent. For every demand request, Pythia observes multiple different types of program context information to make a prefetch decision. For every prefetch decision, Pythia receives a numerical reward that evaluates prefetch quality under the current memory bandwidth usage. Pythia uses this reward to reinforce the correlation between program context information and prefetch decision to generate highly accurate, timely, and system-aware prefetch requests in the future. Our extensive evaluations using simulation and hardware synthesis show that Pythia outperforms two state-of-the-art prefetchers (MLOP and Bingo) by 3.4% and 3.8% in single-core, 7.7% and 9.6% in twelve-core, and 16.9% and 20.2% in bandwidth-constrained core configurations, while incurring only 1.03% area overhead over a desktop-class processor and no software changes in workloads. The source code of Pythia can be freely downloaded from https://github.com/CMU-SAFARI/Pythia.

ACM Reference Format:
Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3466752.3480114

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480114

1 INTRODUCTION

Prefetching is a well-studied speculation technique that predicts the addresses of long-latency memory requests and fetches the corresponding data from main memory to on-chip caches before the program executing on the processor demands it. A program's repeated accesses over its data structures create patterns in its memory request addresses. A prefetcher tries to identify such memory access patterns from past memory requests to predict the addresses of future memory requests. To quickly identify a memory access pattern, a prefetcher typically uses some program context information to examine only a subset of memory requests. We call this program context a *feature*. The prefetcher associates a memory access pattern with a feature and generates prefetches following the same pattern when the feature reoccurs during program execution.

Past research has proposed numerous prefetchers that consistently pushed the limits of prefetch coverage (i.e., the fraction of memory requests predicted by the prefetcher) and accuracy (i.e., the fraction of prefetch requests that are actually demanded by the program) by exploiting various program features, e.g., program counter (`PC`), cacheline address (`Address`), page offset of a cacheline (`Offset`), or a simple combination of such features using simple operations like concatenation (`+`) [25, 27, 30, 32, 35, 53, 55, 56, 65, 73, 78–80, 90, 103, 106, 111, 112, 122, 123]. For example, a PC-based stride prefetcher [55, 56, 73] uses PC as the feature to learn the constant stride between two consecutive memory accesses caused by the same PC. VLDP [112] and SPP [78] use a sequence of cacheline address deltas as the feature to predict the next cacheline address delta. Kumar and Wilkerson [80] use `PC+Address` of the first access in a memory region as the feature to predict the spatial memory access footprint in the entire memory region. SMS [122] empirically finds `PC+Offset` of the first access in a memory region to be a better feature to predict the memory access footprint. Bingo [27] combines the features from [80] and SMS and uses `PC+Address` and `PC+Offset` as its features.

Accurate and timely prefetch requests reduce the long memory access latency experienced by the processor, thereby improving overall system performance. However, speculative prefetch requests can cause undesirable effects on the system (e.g., increased memory bandwidth consumption, cache pollution, memory access interference, etc.), which can reduce or negate the performance improvement gained by hiding memory access latency [48, 123]. Thus, a good prefetcher aims to maximize its benefits while minimizing its undesirable effects on the system.

Even though there is a large number of prefetchers proposed in the literature, we observe three key shortcomings in almost every prior prefetcher design that significantly limits its performance benefits over a wide range of workloads and system configurations:

(1) the use of mainly a single program feature for prefetch prediction, (2) lack of inherent system awareness, and (3) lack of ability to customize the prefetcher design to seamlessly adapt to a wide range of workload and system configurations.

Single-feature prefetch prediction. Almost every prior prefetch-er relies on *only one* program feature to correlate with the program memory access pattern and generate prefetch requests [25, 30, 32, 35, 53, 55, 56, 65, 73, 78–80, 90, 103, 106, 111, 112, 122, 123]. As a result, a prefetcher typically provides good (or poor) performance benefits in mainly those workloads where the correlation between the feature used by the prefetcher and program's memory access pattern is dominantly present (or absent). To demonstrate this, we show the coverage and overpredictions (i.e., prefetched memory requests that do *not* get demanded by the processor) of two recently proposed prefetchers, SPP [78] and Bingo [27], and our new proposal Pythia (§4) for six example workloads (§5 discusses our experimental methodology) in Fig. 1(a). Fig. 1(b) shows the performance of SPP, Bingo and Pythia on the same workloads. As we see in Fig. 1(a), Bingo provides higher prefetch coverage than SPP in `sphinx3`, `PARSEC-Canneal`, and `PARSEC-Facesim`, where the correlation exists between the first access in a memory region and the other accesses in the same region. As a result, Bingo performs better than SPP in these workloads (Fig. 1(b)). In contrast, for workloads like `GemsFDTD` that have regular access patterns within a physical page, SPP's *sequence of deltas* feature provides better coverage and performance than Bingo.

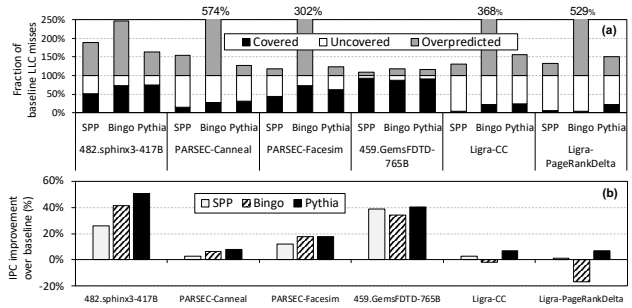

Figure 1: Comparison of (a) coverage, overprediction, and (b) performance of two recently-proposed prefetchers, SPP [78] and Bingo [27], and our new proposal, Pythia.

Lack of inherent system awareness. All prior prefetchers either completely neglect their undesirable effects on the system (e.g., memory bandwidth usage, cache pollution, memory access interference, system energy consumption, etc.) [25, 27, 32, 35, 53, 55, 56, 65, 73, 78–80, 90, 103, 106, 111, 112, 122] or incorporate system awareness as an afterthought (i.e., a separate control component) to the underlying system-unaware prefetch algorithm [30, 34, 47–49, 81, 82, 85, 86, 95, 123, 144]. Due to the lack of inherent system awareness, a prefetcher often loses its performance gain in resource-constrained scenarios. For example, as shown in Fig. 1(a), Bingo achieves similar prefetch coverage in `Ligra-CC` as compared to `PARSEC-Canneal`, while generating significantly lower overpredictions in `Ligra-CC` than `PARSEC-Canneal`. However, Bingo loses performance in `Ligra-CC` by 1.9% compared to a no-prefetching baseline, whereas it improves performance by 6.4% in `PARSEC-Canneal` (Fig. 1(b)). This contrasting outcome is due to Bingo's lack of awareness of the memory bandwidth usage. Without prefetching, `Ligra-CC` consumes higher memory bandwidth than `PARSEC-Canneal`. As a result, each overprediction made by Bingo in `Ligra-CC` wastes more precious memory bandwidth and is more detrimental to performance than that in `PARSEC-Canneal`.

Lack of online prefetcher design customization. The high design complexity of architecting a multi-feature, system-aware prefetcher has traditionally compelled architects to statically select only one program feature at design time. With every new prefetcher, architects design new rigid hardware structures to exploit the selected program feature. To exploit a new program feature for higher performance benefits, one must design a new prefetcher from scratch and extensively evaluate and verify it both in pre-silicon and post-silicon realization. Due to the rigid design-time decisions, the hardware structures proposed by prior prefetchers cannot be customized online in silicon either to exploit any other program feature or to change the prefetcher's objective (e.g., to increase/decrease coverage, accuracy, or timeliness) so that it can seamlessly adapt to varying workloads and system configurations.

Our goal in this work is to design a single prefetching framework that (1) can holistically learn to prefetch using both *multiple different types of program features* and *system-level feedback* information that is inherent to the design, and (2) can be *easily customized* in silicon via simple configuration registers to exploit different types of program features and/or to change the objective of the prefetcher (e.g., increasing/decreasing coverage, accuracy, or timeliness) without any changes to the underlying hardware.

Key ideas. To this end, we propose Pythia,[1] which formulates hardware prefetching as a reinforcement learning problem. Reinforcement learning (RL) [64, 124] is a machine learning paradigm that studies how an autonomous agent can learn to take optimal actions that maximizes a reward function by interacting with a stochastic environment. We formulate Pythia as an RL-agent that autonomously learns to prefetch by interacting with the processor and the memory subsystem. For every new demand request, Pythia extracts a set of program features. It uses the set of features as *state* information to take a prefetch *action* based on its prior experience. For every prefetch action (including *not to prefetch*), Pythia receives a numerical *reward* which evaluates the accuracy and timeliness of the prefetch action given various system-level feedback information. While Pythia's framework is general enough to incorporate any type of system-level feedback information into its decision making, in this paper we demonstrate Pythia using one major system-level information for prefetching: memory bandwidth usage. Pythia uses the reward received for a prefetch action to reinforce the correlations between various program features and the prefetch action and learn from experience how to generate accurate, timely, and system-aware prefetches in the future. The types of program feature used by Pythia and the reward level values can be easily customized in silicon via configuration registers.

Novelty and Benefits. Pythia's RL-based design approach requires an architect to only specify *which* of the possible program features *might* be useful to design a good prefetcher and *what* performance goal the prefetcher should target, rather than spending

[1] Pythia, according to Greek mythology, is the oracle of Delphi who is known for accurate prophecies [18].

time on designing and implementing a new (likely rigid) prefetch algorithm and accompanying rigid hardware that describes *precisely how* the prefetcher should exploit the selected features to achieve that performance goal. This approach provides two unique advantages over prior prefetching proposals. First, using the RL framework, Pythia can holistically learn to prefetch using both *multiple program features* and *system-level feedback* information inherent to its design. Second, Pythia can be easily customized in silicon via simple configuration registers to exploit different types of program features and/or change the objective of the prefetcher. This gives Pythia the unique benefit of providing even higher performance improvements for a wide variety of workloads and changing system configurations, without any changes to the underlying hardware.

Results Summary. We evaluate Pythia using a diverse set of memory-intensive workloads spanning `SPEC CPU2006` [21], `SPEC CPU2017` [22], `PARSEC 2.1` [16], `Ligra` [117], and `Cloudsuite` [51] benchmarks. We demonstrate four key results. First, Pythia outperforms two state-of-the-art prefetchers (MLOP [111] and Bingo [27]) by 3.4% and 3.8% in single-core and 7.7% and 9.6% in twelve-core configurations. This is because Pythia generates lower overpredictions, while simultaneously providing higher prefetch coverage than the prior prefetchers. Second, Pythia's performance benefits increase in bandwidth-constrained system configurations. For example, in a server-like configuration, where a core can have only $\frac{1}{16}\times$ of the bandwidth of a single-channel DDR4-2400 [15] DRAM controller, Pythia outperforms MLOP and Bingo by 16.9% and 20.2%. Third, Pythia can be customized further via simple configuration registers to target workload suites to provide even higher performance benefits. We demonstrate that by simply changing the numerical rewards, Pythia provides up to 7.8% (1.9% on average) more performance improvement across all `Ligra` graph processing workloads over the basic Pythia configuration. Fourth, Pythia's performance benefits come with only modest area and power overheads. Our functionally-verified hardware synthesis for Pythia shows that Pythia only incurs an area and power overhead of 1.03% and 0.37% over a 4-core desktop-class processor.

We make the following contributions in this paper:

- We observe three key shortcomings in prior prefetchers that significantly limits their performance benefits: (1) the use of only a single program feature for prefetch prediction, (2) lack of inherent system awareness, and (3) lack of ability to customize the prefetcher design to seamlessly adapt to a wide range of workloads and system configurations.
- We introduce a new prefetcher called Pythia. Pythia formulates the prefetcher as a reinforcement learning (RL) agent, which takes adaptive prefetch decisions by autonomously learning using both multiple program features and system-level feedback information inherent to its design (§3.1).
- We provide a low-overhead, practical implementation of Pythia's RL-based algorithm in hardware, which uses no more complex structures than simple tables (§4.2.1). This design can potentially be used for other hardware structures that can benefit from RL principles.
- By extensive evaluation, we show that Pythia outperforms prior state-of-the-art prefetchers over a wide variety of workloads in a wide range of system configurations.
- We open source Pythia and all the workload traces used for performance modeling in our GitHub repository: https://github.com/CMU-SAFARI/Pythia.

2 BACKGROUND

We first briefly review the basics of reinforcement learning [64, 124]. We then describe why reinforcement learning is a good framework for designing a hardware prefetcher that fits our goals.

2.1 Reinforcement Learning

Reinforcement learning (RL) [64, 124], in its simplest form, is the algorithmic approach to learn how to take an *action* in a given *situation* to maximize a numerical *reward* signal. A typical RL system comprises of two main components: *the agent* and *the environment*, as shown in Fig. 2. The agent is the entity that takes actions. the agent resides in the environment and interacts with it in discrete timesteps. At each timestep t, the agent observes the current ***state*** of the environment S_t and takes ***action*** A_t. Upon receiving the action, the environment transitions to a new state $\mathbf{S_{t+1}}$, and emits an immediate ***reward*** R_{t+1}, which is immediately or later delivered to the agent. The reward scheme encapsulates the agent's objective and drives the agent towards taking optimal actions.

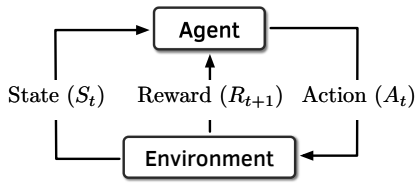

Figure 2: Interaction between an agent and the environment in a reinforcement learning system.

The ***policy*** of the agent dictates it to take a certain action in a given state. *The agent's goal is to find the optimal policy that maximizes the cumulative reward collected from the environment over time.* The expected cumulative reward by taking an action A in a given state S is defined as the ***Q-value*** of the state-action pair (denoted as $Q(S, A)$). At every timestep t, the agent iteratively optimizes its policy in two steps: (1) the agent updates the Q-value of a state-action pair using the reward collected in the current timestep, and (2) the agent optimizes its current policy using the newly updated Q-value.

Updating Q-values. If at a given timestep t, the agent observes a state S_t, takes an action A_t, while the environment transitions to a new state S_{t+1} and emits a reward R_{t+1} and the agent takes action A_{t+1} in the new state, the Q-value of the old state-action pair $Q(S_t, A_t)$ is iteratively optimized using the SARSA [108, 124] algorithm, as shown in Eqn. (1):

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

α is the *learning rate* parameter that controls the convergence rate of Q-values. γ is the *discount factor*, which is used to assign more weight to the immediate reward received by the agent at any given timestep than to the delayed future rewards. A γ value closer to 1 gives a "far-sighted" planning capability to the agent, i.e., the agent can trade off a low immediate reward to gain higher rewards

in the future. This is particularly useful in creating an autonomous agent that can anticipate the long-term effects of taking an action to optimize its policy that gets closer to optimal over time.

Optimizing policy. To find a policy that maximizes the cumulative reward collected over time, a purely-greedy agent always exploits the action A in a given state S that provides the highest Q-value $Q(S, A)$. However, greedy exploitation can leave the state-action space under-explored. Thus, in order to strike a balance between exploration and exploitation, an ϵ-greedy agent *stochastically* takes a random action with a low probability of ϵ (called *exploration rate*); otherwise, it selects the action that provides the highest Q-value [124].

In short, the Q-value serves as the foundational cornerstone of reinforcement learning. By iteratively learning Q-values of state-action pairs, an RL-agent continuously optimizes its policy to take actions that get closer to optimal over time.

2.2 Why is RL a Good Fit for Prefetching?

The RL framework has been recently successfully demonstrated to solve complex problems like mastering human-like control on Atari [92] and Go [118, 119]. We argue that the RL framework is an inherent fit to model a hardware prefetcher for three key reasons.

Adaptive learning in a complex state space. As we state in §1, the benefits of a prefetcher not only depends on its coverage and accuracy but also on its undesirable effects on the system, like memory bandwidth usage. In other words, *it is not sufficient for a prefetcher only to make highly accurate predictions.* Instead, a prefetcher should be *performance-driven.* A prefetcher should have the capability to adaptively trade-off coverage for higher accuracy (and vice-versa) depending on its impact on the overall system to provide a robust performance improvement with varying workloads and system configurations. This adaptive and performance-driven nature of prefetching in a complex state space makes RL a good fit for modeling a prefetcher as an autonomous agent that learns to prefetch by interacting with the system.

Online learning. An RL agent *does not* require an expensive offline training phase. Instead, it can *continuously* learn *online* by iteratively optimizing its policy using the rewards received from the environment. A hardware prefetcher, similar to an RL agent, also needs to continuously learn from the changing workload behavior and system conditions to provide consistent performance benefits. The online learning requirement of prefetching makes RL an inherent fit to model a hardware prefetcher.

Ease of implementation. Prior works have evaluated many sophisticated machine learning models like simple neural networks [105], LSTMs [61, 114], and Graph Neural Networks (GNNs) [116] as models for hardware prefetching. Even though these techniques show encouraging results in accurately predicting memory accesses, they fall short especially in two major aspects. First, these models' sizes often exceed even the largest caches in traditional processors [61, 105, 114, 116], making them impractical (or at best very difficult) to implement. Second, due to the vast amount of computation they require for inference, these models' inference latency is much higher than an acceptable latency of a prefetcher at any cache level. On the other hand, we can efficiently implement an RL-based model, as we demonstrate in this paper (§4), that can *quickly* make predictions and can be relatively easily adopted in a real processor.

3 PYTHIA: KEY IDEA

In this work, we formulate prefetching as a reinforcement learning problem, as shown in Fig. 3. Specifically, we formulate Pythia as an RL-agent that learns to make accurate, timely, and system-aware prefetch decisions by interacting with the environment, i.e., the processor and the memory subsystem. Each timestep corresponds to a new demand request seen by Pythia. With every new demand request, Pythia observes the state of the processor and the memory subsystem and takes a prefetch action. For every prefetch action (including *not to prefetch*), Pythia receives a numerical reward that evaluates the accuracy and timeliness of the prefetch action taking into account various system-level feedback information. *Pythia's goal is to find the optimal prefetching policy that would maximize the number of accurate and timely prefetch requests, taking system-level feedback information into account.* While Pythia's framework is general enough to incorporate any type of system-level feedback into its decision making, in this paper we demonstrate Pythia using *memory bandwidth usage* as the system-level feedback information.

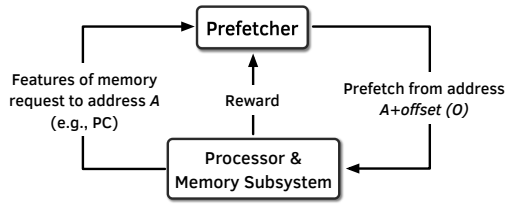

Figure 3: Formulating the prefetcher as an RL-agent.

3.1 Formulation of the RL-based Prefetcher

We formally define the three pillars of our RL-based prefetcher: the state space, the actions, and the reward scheme.

State. We define the state as a k-dimensional vector of program features.

$$S \equiv \{\phi_S^1, \phi_S^2, \ldots, \phi_S^k\} \quad (2)$$

Each program feature is composed of at most two components: (1) program control-flow component, and (2) program data-flow component. The control-flow component is further made up of simple information like load-PC (i.e., the PC of a load instruction) or branch-PC (i.e., the PC of a branch instruction that immediately precedes a load instruction), and a history that denotes whether this information is extracted only from the current demand request or a series of past demand requests. Similarly, the data-flow component is made up of simple information like cacheline address, physical page number, page offset, cacheline delta, and its corresponding history. Table 1 shows some example program features. Although Pythia can theoretically learn to prefetch using *any* number of such program features, we fix the state-vector dimension (i.e., k) at design time given a limited storage budget in hardware. However, the exact selection of k program features out of all possible program features is configurable online using simple configuration registers. In §4.3.1, we provide an *automated feature selection method* to find a vector of program features to be used at design time.

Action. We define the action of the RL-agent as selecting a *prefetch offset* (i.e., a delta, "O" in Fig. 3, between the predicted and

Table 1: Example program features

Feature	Control-flow		Data-flow	
	Info.	History	Info.	History
Last 3-PCs	PC	last 3	✖	✖
Last 4-deltas	✖	✖	Cacheline delta	last 4
PC+Delta	PC	current	Cacheline delta	current
Last 4-PCs+Page no.	PC	last 4	Page no.	current

the demanded cacheline address) from a set of candidate prefetch offsets. As every post-L1-cache prefetcher generates prefetch requests within a physical page [27, 30, 32, 53, 65, 78–80, 90, 103, 106, 111, 112, 122, 123], the list of prefetch offsets only contains values in the range of $[-63, 63]$ for a system with a traditionally-sized 4KB page and 64B cacheline. Using prefetch offsets as actions (instead of full cacheline addresses) drastically reduces the action space size. We further reduce the action space size by fine tuning, as described in §4.3.2. A prefetch offset of zero means no prefetch is generated.

Reward. The reward structure defines the prefetcher's objective. We define five different reward levels as follows.

- ***Accurate and timely*** ($\mathcal{R}_{AT}$). This reward is assigned to an action whose corresponding prefetch address gets demanded *after* the prefetch fill.
- ***Accurate but late*** ($\mathcal{R}_{AL}$). This reward is assigned to an action whose corresponding prefetch address gets demanded *before* the prefetch fill.
- ***Loss of coverage*** ($\mathcal{R}_{CL}$). This reward is assigned to an action whose corresponding prefetch address is to a different physical page than the demand access that led to the prefetch.
- ***Inaccurate*** ($\mathcal{R}_{IN}$). This reward is assigned to an action whose corresponding prefetch address does *not* get demanded in a temporal window. The reward is classified into two sub-levels: inaccurate given low bandwidth usage ($\mathcal{R}_{IN}^{L}$) and inaccurate given high bandwidth usage ($\mathcal{R}_{IN}^{H}$).
- ***No-prefetch*** ($\mathcal{R}_{NP}$). This reward is assigned when Pythia decides not to prefetch. This reward level is also classified into two sub-levels: no-prefetch given low bandwidth usage ($\mathcal{R}_{NP}^{L}$) and no-prefetch given high bandwidth usage ($\mathcal{R}_{NP}^{H}$).

By increasing (decreasing) a reward level value, we reinforce (deter) Pythia to collect such rewards from the environment in the future. $\mathcal{R}_{AT}$ and $\mathcal{R}_{AL}$ are used to guide Pythia to generate more accurate and timely prefetch requests. $\mathcal{R}_{CL}$ is used to guide Pythia to generate prefetches within the physical page of the triggering demand request. $\mathcal{R}_{IN}$ and $\mathcal{R}_{NP}$ are used to define Pythia's prefetching strategy with respect to memory bandwidth usage feedback. In §4.3.3, we provide an *automated method* to configure the reward values. The reward values can be easily customized further for target workload suites to extract higher performance gains (§6.6).

4 PYTHIA: DESIGN

Fig. 4 shows a high-level overview of Pythia. Pythia is mainly comprised of two hardware structures: *Q-Value Store* (QVStore) and *Evaluation Queue* (EQ). The purpose of QVStore is to record Q-values for all state-action pairs that are observed by Pythia. The purpose of EQ is to maintain a first-in-first-out list of Pythia's recently-taken actions.[2] Every EQ entry holds three pieces of information: (1) the taken action, (2) the prefetch address generated for the corresponding action, and (3) a *filled* bit. A set filled bit indicates that the prefetch request has been filled into the cache.

For every new demand request, Pythia first checks the EQ with the demanded memory address (❶). If the address is present in the EQ (i.e., Pythia has issued a prefetch request for this address in the past), it signifies that the prefetch action corresponding to the EQ entry has generated a useful prefetch request. As such, Pythia assigns a reward (either $\mathcal{R}_{AT}$ or $\mathcal{R}_{AL}$) to the EQ entry, based on whether or not the EQ entry's filled bit is set. Next, Pythia extracts the state-vector from the attributes of the demand request (e.g., PC, address, cacheline delta, etc.) (❷) and looks up QVStore to find the action with the maximum Q-value for the given state-vector (❸). Pythia selects the action with the maximum Q-value to generate prefetch request and issues the request to the memory hierarchy (❹). At the same time, Pythia inserts the selected prefetch action, its corresponding prefetched memory address, and the state-vector into EQ (❺). Note that, a *no-prefetch* action or an action that prefetches an address beyond the current physical page is also inserted into EQ. The reward for such an action is instantaneously assigned to the EQ entry. When an EQ entry gets evicted, the state-action pair and the reward stored in the evicted EQ entry are used to update the Q-value in the QVStore (❻). For every prefetch fill in cache, Pythia looks up EQ with the prefetch address and sets the *filled* bit in the matching EQ entry indicating that the prefetch request has been filled into the cache (❼). Pythia uses this filled bit in ❶ to classify actions that generated timely or late prefetches.[3]

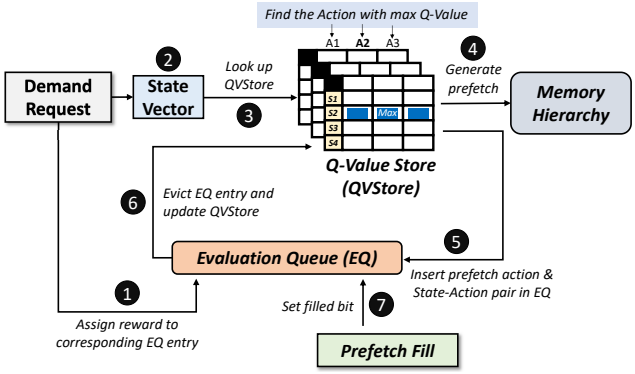

Figure 4: Overview of Pythia.

4.1 RL-based Prefetching Algorithm

Algorithm 1 shows Pythia's RL-based prefetching algorithm. Initially, all entries in QVStore are reset to the highest possible Q-value ($\frac{1}{1-\gamma}$) and the EQ is cleared (lines 2-3). For every demand request to a cacheline address *Addr*, Pythia searches for *Addr* in EQ (line 6). If a matching entry is found, Pythia assigns a reward (either $\mathcal{R}_{AT}$ or

[2] Pythia keeps track of recently-taken actions because it cannot always *immediately* assign a reward to an action, as the usefulness of the generated prefetch request (i.e., if and when the prefetched address is demanded by the processor) is not immediately known while the action is being taken. During EQ residency, if the address of a demand request matches with the prefetch address stored in an EQ entry, the corresponding action is considered to have generated a useful prefetch request.

[3] In this paper, we define prefetch timeliness as a binary value due to its measurement simplicity. One can easily make the definition non-binary by storing three timestamps per EQ entry: (1) when the prefetch is issued (t_{issue}), (2) when the prefetch is filled (t_{fill}), and (3) when a demand is generated for the same prefetched address (t_{demand}).

Algorithm 1 Pythia's reinforcement learning based prefetching algorithm

```
1:  procedure INITIALIZE
2:      initialize QVStore: Q(S, A) ← 1/(1−γ)
3:      clear EQ
4:
5:  procedure TRAIN_AND_PREDICT(Addr)          /* Called for every demand request */
6:      entry ← search_EQ(Addr)                /* For a demand request to Addr, search EQ with the demand address */
7:      if entry is valid then
8:          if entry.filled == true then
9:              entry.reward ← R_AT            /* If the filled bit is set, i.e., the demand access came after the prefetch fill, assign reward R_AT */
10:         else
11:             entry.reward ← R_AL            /* Otherwise, assign R_AL */
12:     S ← get_state()                        /* Extract the state-vector from the attributes of current demand request */
13:     if rand() ≤ ϵ then
14:         action ← get_random_action()       /* Select a random action with a low probability ϵ to explore the state-action space */
15:     else
16:         action ← argmax_a Q(S, a)          /* Otherwise, select the action with the highest Q-value */
17:     prefetch(Addr + Offset[action])        /* Add the selected prefetch offset to the current demand address to generate prefetch address */
18:     entry ← create_EQ_entry(S, action, Addr + Offset[action])   /* Create new EQ entry using the current state-vector, the selected action, and the prefetch address */
19:     if no prefetch action then
20:         entry.reward ← R^H_NP or R^L_NP    /* In case of no-prefetch action, immediately assign reward R^H_NP or R^L_NP based on current memory bandwidth usage */
21:     else if out-of-page prefetch then
22:         entry.reward ← R_CL                /* In case of out-of-page prefetch action, immediately assign reward R_CL */
23:     evict_entry ← insert_EQ(entry)         /* Insert the entry. Get the evicted EQ entry. */
24:     if has_reward(dq_entry) == false then
25:         dq_entry.reward ← R^H_IN or R^L_IN /* If the evicted entry does not have a reward yet, assign the reward R^H_IN or R^L_IN based on current memory bandwidth usage */
26:     R ← dq_entry.reward                    /* Get the reward stored in the evicted entry */
27:     S_1 ← dq_entry.state;  A_1 ← dq_entry.action   /* Get the state-vector and the action from the evicted EQ entry */
28:     S_2 ← EQ.head.state;  A_2 ← EQ.head.action     /* Get the state-vector and the action from the entry at the head of the EQ */
29:     Q(S_1, A_1) ← Q(S_1, A_1) + α[R + γQ(S_2, A_2) − Q(S_1, A_1)]   /* Perform the SARSA update */
30:
31: procedure PREFETCH_FILL(Addr)
32:     search_and_mark_EQ(Addr, FILLED)       /* For every prefetch fill, search the address in EQ and mark the corresponding EQ entry as filled */
```

$\mathcal{R}_{AL}$) based on the *filled* bit in the EQ entry (lines 8-11). Pythia then extracts the state-vector to *stochastically* select a prefetching action (Sec. 2) that provides the highest Q-value (lines 13-16). Pythia uses the selected action to generate the prefetch request (line 17) and creates a new EQ entry with the current state-vector, the selected action, and its corresponding prefetched address (line 18). In case of a no-prefetch action, or an action that prefetches beyond the current physical page, Pythia immediately assigns the reward to the newly-created EQ entry (lines 19-22). The EQ entry is then inserted, which evicts an entry from EQ. If the evicted EQ entry does not already have a reward assigned (indicating that the corresponding prefetch address is *not* demanded by the processor so far), Pythia assigns the reward $\mathcal{R}^H_{IN}$ or $\mathcal{R}^L_{IN}$ based on the current memory bandwidth usage (lines 25). Finally, the Q-value of the evicted state-action pair is updated via the SARSA algorithm (Sec. 2), using the reward stored in the evicted EQ entry and the Q-value of the state-action pair in the head of the EQ-entry (lines 26-29).

4.2 Detailed Design of Pythia

We describe the organization of QVStore (§ 4.2.1), how Pythia searches QVStore to get the action with the maximum Q-value for a given state-vector (❸) (§4.2.2), how Pythia assigns rewards to each taken action and how it updates Q-values (❻) (§4.2.3).

4.2.1 ***Organization of QVStore.*** The purpose of QVStore is to record Q-values for all state-action pairs that Pythia observes. Unlike prior real-world applications of RL [92, 118, 119], which use deep neural networks to *approximately* store Q-values of every state-action pair, we propose a new, table-based, hierarchical QVStore organization that is custom-designed to our RL-agent.

Fig. 5(a) shows the high-level organization of QVStore and how the Q-value is retrieved from QVStore for a given state S (which is a k-dimensional vector of program features, $\{\phi^1_S, \phi^2_S, \ldots, \phi^k_S\}$) and an action A. As the state space grows rapidly with the state-vector dimension (k) and the bits used to represent each feature, we employ a hierarchical organization for QVStore. We organize QVStore in k partitions, each of which we call a *vault*. Each vault corresponds to one constituent feature of the state-vector and records the Q-values for the feature-action pair, $Q(\phi^i_S, A)$. During the Q-value retrieval for a given state-action pair $Q(S, A)$, Pythia queries each vault in parallel to retrieve the Q-values of constituent feature-action pairs $Q(\phi^i_S, A)$. The final Q-value of the state-action pair $Q(S, A)$ is computed as the *maximum* of all constituent feature-action Q-values, as Eqn. 3 shows). The maximum operation ensures that the state-action Q-value is driven by the constituent feature of the state-vector that has the highest feature-action Q-value. The vault organization enables QVStore to efficiently scale up to higher state-vector dimensions: one can increase the state-vector dimension by simply adding a new vault to the QVStore.

$$Q(S, A) = \max_{i \in (1,k)} Q(\phi^i_S, A) \tag{3}$$

Fig. 5(a) shows the organization of QVStore as a collection of multiple vaults. The purpose of a vault is to record Q-values of all feature-action pairs that Pythia observes for a specific feature type. A vault can be conceptually visualized as a monolithic two-dimensional table (as shown in Fig. 5(a)), indexed by the feature

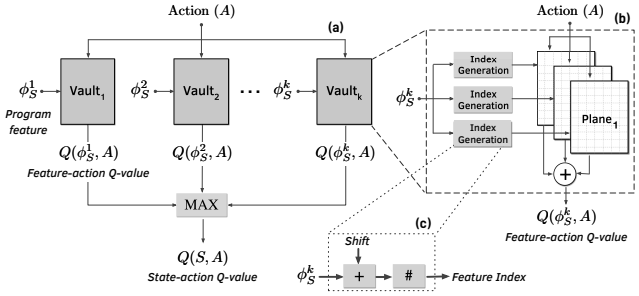

Figure 5: (a) The QVStore is comprised of multiple vaults. (b) Each vault is comprised of multiple planes. (c) Index generation from feature value.

and action values, that stores Q-value for every feature-action pair. However, the key challenge in implementing vault as a monolithic table is that the size of the table increases exponentially with a linear increase in the number of bits used to represent the feature. This not only makes the monolithic table organization impractical for implementation but also increases the design complexity to satisfy its latency and power requirements.

One way to address this challenge is to quantize the feature space into a small number of tiles. Even though feature space quantization can achieve a drastic reduction in the monolithic table size, it requires a compromise between the resolution of a feature value and the generalization of feature values. We draw inspiration from *tile coding* [24, 64, 124] to strike a balance between resolution and generalization. Tile coding uses *multiple overlapping* hash functions to quantize a feature value into smaller *tiles*. The quantization achieves generalization of similar feature values, whereas multiple hash functions increase resolution to represent a feature value.

We leverage the idea of tile coding to organize a vault as a collection of N small two-dimensional tables, each of which we call a *plane*. Each plane entry stores a *partial* Q-value of a feature-action pair.[4] As Fig. 5(c) shows, to retrieve a feature-action Q-value $Q(\phi_S^i, A)$, the given feature is first shifted by a shifting constant (which is randomly selected at design time), followed by a hashing to get the feature index for the given plane. This feature index, along with the action index, is used to retrieve the partial Q-value from the plane. The final feature-action Q-value is computed as the *sum of all* the partial Q-values from all planes, as shown in Fig. 5(b). The use of tile coding provides two key advantages to Pythia. First, the tile coding of a feature enables the sharing of partial Q-values between similar feature values, which shortens prefetcher training time. Second, multiple planes reduces the chance of sharing partial Q-values between widely different feature values.

4.2.2 ***Pipelined Organization of QVStore Search.*** To generate a prefetch request, Pythia has to (1) look up the QVStore with the state-vector extracted from the current demand request, and (2) search for the action that has the maximum state-action Q-value (❸ in Fig. 4). As a result, the search operation lies on Pythia's critical path and directly impacts Pythia's prediction latency. To improve the prediction latency, we pipeline the search operation.

[4]Our application of tile coding is similar to that used in the self-optimizing memory controller (RLMC) [64]. The key difference is that RLMC uses a hybrid combination of feature and action values to index *single-dimensional* planes, whereas Pythia uses feature and action values *separately* to index *two-dimensional* planes.

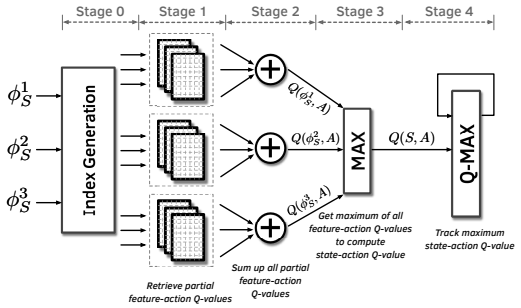

Figure 6: Pipelined organization of QVStore search operation. The illustration depicts three program features, each having three planes.

The Q-value search operation is implemented in the following way. For a given state-vector, Pythia iteratively retrieves the Q-value of each action. Pythia also maintains a variable, Q_{max}, that tracks the maximum Q-value found so far. Q_{max} gets compared to every retrieved Q-value. The search operation concludes when Q-values for *all* possible actions have been retrieved. We pipeline the search operation into five stages as Fig. 6 shows. Pythia first computes the index for each plane and each constituent feature of the given state-vector (Stage 0). In Stage 1, Pythia uses the feature indices and an action index to retrieve the partial Q-values from each plane. In Stage 2, Pythia sums up the partial Q-values to get the feature-action Q-value for each constituent feature. In Stage 3, Pythia computes the maximum of all feature-action Q-values to get the state-action Q-value. In Stage 4, the maximum state-action Q-value found so far is compared against the retrieved state-action Q-value, and the maximum Q-value is updated. Stage 2 (i.e., the partial Q-value summation) is the longest stage of the pipeline and thus it dictates the pipeline's throughput. We accurately measure the area and power overhead of the pipelined implementation of the search operation by modeling Pythia using Chisel [8] hardware design language and synthesize the model using Synopsys design compiler [23] and 14-nm library from GlobalFoundries [10] (§6.7).

4.2.3 ***Assigning Rewards and Updating Q-values.*** To track usefulness of the prefetched requests, Pythia maintains a first-in-first-out list of recently taken actions, along with their corresponding prefetch addresses in EQ. *Every* prefetch action is inserted into EQ. A reward gets assigned to every EQ entry before or when it gets evicted from EQ. During eviction, the reward and the state-action pair associated with the evicted EQ entry are used to update the corresponding Q-value in QVStore (❻ in Fig. 4).

We describe how Pythia appropriately assigns rewards to each EQ entry. We divide the reward assignment into three classes based on *when* the reward gets assigned to an entry: (1) immediate reward assignment during EQ insertion, (2) reward assignment during EQ residency, and (3) reward assignment during EQ eviction. If Pythia selects the action *not to prefetch* or one that generates a prefetch request beyond the current physical page, Pythia immediately assigns a reward to the EQ entry. For out-of-page prefetch action, Pythia assigns $\mathcal{R}_{CL}$. For the action *not to prefetch*, Pythia assigns $\mathcal{R}_{NP}^H$ or $\mathcal{R}_{NP}^L$, based on whether the current system memory bandwidth usage is high or low. If the address of a demand request matches with the prefetch address stored in an EQ entry during its residency, Pythia assigns $\mathcal{R}_{AT}$ or $\mathcal{R}_{AL}$ based on the *filled* bit of the EQ entry. If

the filled bit is set, it indicates that the demand request is generated *after* the prefetch fill. Hence the prefetch is accurate and timely, and Pythia assigns the reward $\mathcal{R}_{AT}$. Otherwise, Pythia assigns the reward $\mathcal{R}_{AL}$. If a reward does not get assigned to an EQ entry until it is going to be evicted, it signifies that the corresponding prefetch address is not yet demanded by the processor. Thus, Pythia assigns a reward $\mathcal{R}^H_{IN}$ or $\mathcal{R}^L_{IN}$ to the entry during eviction based on whether the current system memory bandwidth usage is high or low.

4.3 Automated Design-Space Exploration

We propose an automated, performance-driven approach to systematically explore Pythia's vast design space and derive a basic configuration[5] with appropriate program features, action set, reward and hyperparameters. Table 2 shows the basic configuration.

Table 2: Basic Pythia configuration derived from our automated design-space exploration

Features	`PC+Delta`, `Sequence of last-4 deltas`
Prefetch Action List	{-6,-3,-1,0,1,3,4,5,10,11,12,16,22,23,30,32}
Reward Level Values	$\mathcal{R}_{AT}$=20, $\mathcal{R}_{AL}$=12, $\mathcal{R}_{CL}$=−12, $\mathcal{R}^H_{IN}$=−14, $\mathcal{R}^L_{IN}$=−8, $\mathcal{R}^H_{NP}$=−2, $\mathcal{R}^L_{NP}$=−4
Hyperparameters	$\alpha = 0.0065$, $\gamma = 0.556$, $\epsilon = 0.002$

4.3.1 ***Feature Selection.*** We derive a list of possible program features for feature-space exploration in four steps. First, we derive a list of 4 control-flow components, and 8 data-flow components, which are mentioned in Table 3. Second, we combine each control-flow component with each data-flow component with the concatenation operation, to obtain a total of 32 possible program features. Third, we use the linear regression technique [58, 93, 109] to create any-one, any-two, and any-three feature-combinations from the set of 32 initial features, each providing a different state-vector. Fourth, we run Pythia with every state-vectors across all single-core workloads (§5) and select the winning state-vector that provides the highest performance gain over no-prefetching baseline. As Table 2 shows, the two constituent features of the winning state-vector are `PC+Delta` and `Sequence of last-4 deltas`.

Table 3: List of program control-flow and data-flow components used to derive the list of features for exploration

Control-flow Component	Data-flow Component
(1) PC of load request (2) PC-path (XOR-ed last-3 PCs) (3) PC XOR-ed branch-PC (4) None	(1) Load cacheline address (2) Page number (3) Page offset (4) Load address delta (5) Sequence of last-4 offsets (6) Sequence of last-4 deltas (7) Offset XOR-ed with delta (8) None

Rationale behind the winning state-vector. The winning state-vector is intuitive as its constituent features `PC+Delta` and `Sequence of last-4 deltas` closely match with the program features exploited by two prior state-of-the-art prefetchers, Bingo [27] and SPP [78], respectively. However, concurrently running SPP and Bingo as a hybrid prefetcher does not provide the same performance benefit as Pythia, as we show in §6.3.1. This is because combining SPP with Bingo not only improves their prefetch coverage, but also combines their prefetch overpredictions, leading to performance degradation, especially in resource-constrained systems. In contrast, Pythia's RL-based learning strategy that inherently uses the same two features successfully increases prefetch coverage, while maintaining high prefetch accuracy. As a result, *Pythia not only outperforms SPP and Bingo individually, but also outperforms the combination of the two prefetchers.*

4.3.2 ***Action Selection.*** In a system with conventionally-sized 4KB pages and 64B cachelines, Pythia's list of actions (i.e., the list of possible prefetch offsets) contains *all* prefetch offsets in the range of $[-63, 63]$. However, such a long action list poses two drawbacks. First, a long action list requires more online exploration to find the best prefetch offset given a state-vector, thereby reducing Pythia's performance benefits. Second, a longer action list increases Pythia's storage requirements. To avoid these problems, we prune the action list. We drop each action individually from the full action list $[-63, 63]$ and measure the performance improvement relative to the performance improvement with the full action list, across all single-core workload traces. We prune any action that *does not* have any significant impact on the performance. Table 2 shows the final pruned action list.

4.3.3 ***Reward and Hyperparameter Tuning.*** We separately tune seven reward level values (i.e., $\mathcal{R}_{AT}$, $\mathcal{R}_{AL}$, $\mathcal{R}_{CL}$, $\mathcal{R}^H_{IN}$, $\mathcal{R}^L_{IN}$, $\mathcal{R}^H_{NP}$, and $\mathcal{R}^L_{NP}$) and three hyperparameters (i.e., learning rate α, discount factor γ, and exploration rate ϵ) in three steps. First, we create a test trace suite by *randomly* selecting 10 workload traces from all of our 150 workload traces (§5). Second, we create a list of tuning configurations using the uniform grid search technique [31, 83]. To do so, we first define a value range for each parameter to be tuned and divide the value range into uniform grids. For example, each of the three hyperparameters (α, γ, and ϵ) can take a value in the range of $[0, 1]$. We divide each hyperparameter range into ten exponentially-sized grids (i.e., $1e^0$, $1e^{-1}$, $1e^{-2}$, etc.) to obtain $10 \times 10 \times 10 = 1000$ possible tuning configurations. For each tuning configuration, we run Pythia on the test trace suite and select the top-25 highest-performing configurations for the third step. Third, we run Pythia on all single-core workload traces using each of the 25 selected configurations. We select the winning configuration that provides the highest average performance gain. Table 2 provides reward level and hyperparameter values of the basic Pythia.

4.4 Storage Overhead

Table 4 shows the storage overhead of Pythia in its basic configuration. Pythia requires only 25.5 of metadata storage. QVStore consumes 24KB to store all Q-values. The EQ consumes only 1.5KB.

4.5 Differences from Prior Work

The key idea of using RL in prefetching has been previously explored by the *context prefetcher* (CP) [104]. Pythia significantly differs from it both in terms of (1) design principles (i.e., the reward, state, and action definition) and (2) the implementation.

Reward. CP naively defines the agent's reward as a continuous function of prefetch timeliness. Pythia not only considers coverage,

[5] Using a compute-grid with ten 28-core machines, the automated exploration across 150 workload traces (mentioned in detail in §5) takes 44 hours to complete.

Table 4: Storage overhead of Pythia

Structure	Description	Size
QVStore	• # vaults = 2 • # planes in each vault = 3 • # entries in each plane = feature dimension (128) × action dimension (16) • Entry size = Q-value width (16b)	24 KB
EQ	• # entries = 256 • Entry size = state (21b) + action index (5b) + reward (5b) + filled-bit (1b) + address (16b)	1.5 KB
Total		25.5 KB

accuracy, and timeliness but also system-level feedback like memory bandwidth usage to define discrete reward levels. This reward definition provides two key advantages to Pythia. First, unlike CP, Pythia can adaptively trade off prefetch coverage for accuracy (and vice versa) based on memory bandwidth usage. Second, one can easily customize Pythia's objective by changing the reward values via configuration registers to extract even higher performance.

State. CP relies on compiler-generated hints in its state information. In contrast, Pythia extracts program features purely from hardware (e.g., PC, cacheline delta). Thus, Pythia requires no changes to software and it is easier to adopt into existing microprocessors.

Action. Unlike Pythia, CP uses a full cacheline address as the agent's action. The use of full cacheline address as action dramatically increases the action space, which results higher storage cost, longer training time, and reduced performance benefits.

Implementation. Pythia's implementation differs largely from CP in two major ways. First, CP uses the contextual-bandit (CB) algorithm [38], a simplified version of RL. The key difference between CB and RL is that a CB-solver cannot take its actions' *long-term* consequences into account when selecting an action. In contrast, RL-based Pythia weighs each probable prefetch action not only based on the expected immediate reward but also its long-term consequences (e.g., increased bandwidth usage or reduced prefetch accuracy in future) [124]. As such, Pythia provides more robust and far-sighted predictions than the myopic CB-based CP. Second, Pythia organizes the Q-value storage into multiple vaults, each consisting of multiple planes. This hierarchical QVStore structure (1) enables pipelining the Q-value lookup to achieve high-throughput and low-latency prediction, and (2) easily scales out to support longer state-vectors by simply adding more vaults.

5 METHODOLOGY

We use the trace-driven ChampSim simulator [7] to evaluate Pythia and compare it to five prior prefetching proposals. We simulate an Intel Skylake [4]-like multi-core processor that supports up to 12 cores. Table 5 provides the key system parameters. For single-core simulations (1*C*), we warm up the core using 100 M instructions from each workload and simulate the next 500 M instructions. For multi-core multi-programmed simulations (*nC*), we use 50 M and 150 M instructions from each workload respectively to warmup and simulate. If a core finishes early, the workload is replayed until every core finishes simulating 150 M instructions. We also implement Pythia using the Chisel [8] hardware design language (HDL) and functionally verify the resultant register transfer logic (RTL) design to accurately measure Pythia's chip area and power overhead. The source-code of Pythia is freely available at [19].

Table 5: Simulated system parameters

Core	1-12 cores, 4-wide OoO, 256-entry ROB, 72/56-entry LQ/SQ
Branch Pred.	Perceptron-based [69], 20-cycle misprediction penalty
L1/L2 Caches	Private, 32KB/256KB, 64B line, 8 way, LRU, 16/32 MSHRs, 4-cycle/14-cycle round-trip latency
LLC	2MB/core, 64B line, 16 way, SHiP [133], 64 MSHRs per LLC Bank, 34-cycle round-trip latency
Main Memory	**1C:** Single channel, 1 rank/channel; **4C:** Dual channel, 2 ranks/channel; **8C:** Quad channel, 2 ranks/channel; 8 banks/rank, 2400 MTPS, 64b data-bus/channel, 2KB row buffer/bank, tRCD=15ns, tRP=15ns, tCAS=12.5ns

5.1 Workloads

We evaluate Pythia using a diverse set of memory-intensive workloads spanning `SPEC CPU2006` [21], `SPEC CPU2017` [22], `PARSEC 2.1` [16], `Ligra` [117], and `Cloudsuite` [51] benchmark suites. For `SPEC CPU2006` and `SPEC CPU20017` workloads, we reuse the instruction traces provided by the 2nd and the 3rd data prefetching championships (DPC) [2, 3]. For `PARSEC` and `Ligra` workloads, we collect the instruction traces using the Intel Pin dynamic binary instrumentation tool [17]. We do not consider workload traces that have lower than 3 last-level cache misses per kilo instructions (MPKI) in the baseline system with no prefetching. In all, we present results for 150 workload traces spanning 50 workloads. Table 6 shows a categorized view of all the workloads evaluated in this paper. For multi-core multi-programmed simulations, we create both homogeneous and heterogeneous trace mixes from our single-core trace list. For an *n*-core homogeneous trace mix, we run *n* copies of a trace from our single-core trace list, one in each core. For a heterogeneous trace mix, we *randomly* select *n* traces from our single-core trace list and run one trace in every core. All the single-core traces and multi-programmed trace mixes used in our evaluation are freely available online [19].

Table 6: Workloads used for evaluation

Suite	# Workloads	# Traces	Example Workloads
SPEC06	16	28	gcc, mcf, cactusADM, lbm, ...
SPEC17	12	18	gcc, mcf, pop2, fotonik3d, ...
PARSEC	5	11	canneal, facesim, raytrace, ...
Ligra	13	40	BFS, PageRank, Bellman-ford, ...
Cloudsuite	4	53	cassandra, cloud9, nutch, ...

5.2 Prefetchers

We compare Pythia to five state-of-the-art prior prefetchers: SPP [78], SPP+PPF [32], SPP+DSPatch [30], Bingo [27], and MLOP [111]. We model each competing prefetcher using the source-code provided by their respective authors and fine-tune them in our environment to extract the highest performance gain across all single-core traces. Table 7 shows the parameters of all evaluated prefetchers. Each prefetcher is trained on L1-cache misses and fills prefetched lines into L2 and LLC.

We also compare Pythia against multi-level prefetchers found in commercial processors (e.g., stride prefetcher at L1-cache and streamer at L2 [9]) and IPCP [103] in §6.2.4. For fair comparison, we add a simple PC-based stride prefetcher [55, 56, 73] at the L1 level, along with Pythia at the L2 level for such multi-level comparisons.

Table 7: Configuration of evaluated prefetchers

SPP [78]	256-entry ST, 512-entry 4-way PT, 8-entry GHR	**6.2 KB**
Bingo [27]	2KB region, 64/128/4K-entry FT/AT/PHT	**46 KB**
MLOP [111]	128-entry AMT, 500-update, 16-degree	**8 KB**
DSPatch [30]	Same configuration as in [30]	**3.6 KB**
PPF [32]	Same configuration as in [32]	**39.3 KB**
Pythia	2 features, 2 vaults, 3 planes, 16 actions	**25.5 KB**

6 RESULTS

6.1 Coverage and Overprediction in Single-core

Fig. 7 shows the coverage and overprediction of each prefetcher in the single-core system, as measured at the LLC-main memory boundary. The key takeaway is that Pythia improves prefetch coverage, while simultaneously reducing overprediction compared to state-of-the-art prefetchers. On average, Pythia provides 6.9%, 8.8%, and 14% higher coverage than MLOP, Bingo, and SPP respectively, while generating 83.8%, 78.2%, and 3.6% fewer overpredictions.

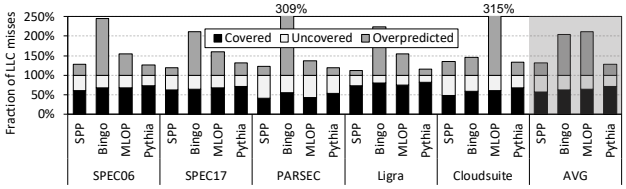

Figure 7: Coverage and overprediction with respect to the baseline LLC misses in the single-core system.

6.2 Performance Overview

6.2.1 ***Varying Number of Cores.*** Figure 8(a) shows the performance improvement of all prefetchers averaged across all traces in single-core to 12-core systems. To realistically model modern commercial multi-core processors, we simulate 1-2 core, 4-6 core, and 8-12 core systems with one, two, and four DDR4-2400 DRAM [15] channels, respectively. We make two key observations from Figure 8(a). First, Pythia consistently outperforms MLOP, Bingo, and SPP in *all* system configurations. Second, Pythia's performance improvement over prior prefetchers increases as core count increases. In the single-core system, Pythia outperforms MLOP, Bingo, SPP, and an aggressive SPP with perceptron filtering (PPF [32]) by 3.4%, 3.8%, 4.3%, and 1.02% respectively. In four (and twelve) core systems, Pythia outperforms MLOP, Bingo, SPP, and SPP+PPF by 5.8% (7.7%), 8.2% (9.6%), 6.5% (6.9%), and 3.1% (5.2%), respectively.

6.2.2 ***Varying DRAM Bandwidth.*** To evaluate Pythia in bandwidth-constrained, highly-multi-threaded commercial server-class processors, where each core can have only a fraction of a channel's bandwidth, we simulate the single-core single-channel configuration by scaling the DRAM bandwidth (Figure 8(b)). Each bandwidth configuration roughly corresponds to the available per-core DRAM bandwidth in various commercial processors (e.g., Intel Xeon Gold [13], AMD EPYC Rome [6], and AMD Threadripper [5]). The key takeaway is that Pythia *consistently* outperforms all competing prefetchers in *every* DRAM bandwidth configuration from $\frac{1}{16}\times$ to 4× bandwidth of the baseline system. Due to their large overprediction rates, the performance gains of MLOP and Bingo reduce sharply as DRAM bandwidth decreases. By actively trading off prefetch coverage for higher accuracy based on memory bandwidth usage, Pythia outperforms MLOP, Bingo, SPP, and SPP+PPF by 16.9%, 20.2%, 3.7%, and 9.5% respectively in the most bandwidth-constrained configuration with 150 million transfers per second (MTPS). In the 9600-MTPS configuration, every prefetcher enjoys ample DRAM bandwidth. Pythia still outperforms MLOP, Bingo, SPP, and SPP+PPF by 3%, 2.7%, 4.4%, and 0.8%, respectively.

6.2.3 ***Varying LLC Size.*** Fig. 8(c) shows performance of all prefetchers averaged across all traces in the single-core system while varying the LLC size from $\frac{1}{8}\times$ to 2× of the baseline 2MB LLC. The key takeaway is that Pythia consistently outperforms all prefetchers in *every* LLC size configuration. For 256KB (and 4MB) LLC, Pythia outperforms MLOP, Bingo, SPP, and SPP+PPF by 3.6% (3.1%), 5.1% (3.4%), 2.7% (4.8%), and 1.2% (0.8%), respectively.

6.2.4 ***Comparison to Multi-level Prefetching Schemes.*** Figure 8(d) shows the performance comparison of Pythia in single-core system with varying DRAM bandwidth against two state-of-the-art *multi-level* prefetching schemes: (1) stride prefetcher [55, 56, 73] at L1 and streamer [35] at L2 cache found in commercial Intel processors [9], and (2) IPCP, the winner of the third data prefetching championship [3]. For fair comparison, we add a stride prefetcher in the L1 cache along with Pythia in the L2 cache for this experiment and measure performance over the no prefetching baseline. The key takeaway is that Stride+Pythia consistently outperforms stride+streamer and IPCP in *every* DRAM bandwidth configuration. Stride+Pythia outperforms Stride+Streamer and IPCP by 6.5% and 14.2% in the 150-MTPS configuration and by 2.3% and 1.0% in the 9600-MTPS configuration, respectively.

6.3 Performance Analysis

6.3.1 ***Single-core.*** Fig. 9(a) shows the performance improvement of each individual prefetcher in each workload category in the single-core system. We make two major observations. First, Pythia improves performance by 22.4% on average over a no-prefetching baseline. Pythia outperforms MLOP, Bingo, and SPP by 3.4%, 3.8%, and 4.3% on average, respectively. Second, only Bingo outperforms Pythia only in the PARSEC suite, by 2.3%. However, Bingo's performance comes at the cost of a high overprediction rate, which hurts performance in multi-core systems (see §6.3.2).

To demonstrate the novelty of Pythia's RL-based prefetching approach using multiple program features, Fig. 9(b) compares Pythia's performance improvement with the performance improvement of various combinations of prior prefetchers. Pythia not only outperforms all prefetchers (stride, SPP, Bingo, DSPatch, and MLOP) individually, but also outperforms their combination by 1.4% on average, with less than half of the combined storage size of the five prefetchers. We conclude that Pythia's RL-based prefetching approach using multiple program features under one single framework provides higher performance benefit than combining multiple prefetchers, each exploiting only one program feature.

6.3.2 ***Four-core.*** Fig. 10(a) shows the performance improvement of each individual prefetcher in each workload category in the four-core system. We make two major observations. First, Pythia provides significant performance improvement over all prefetchers

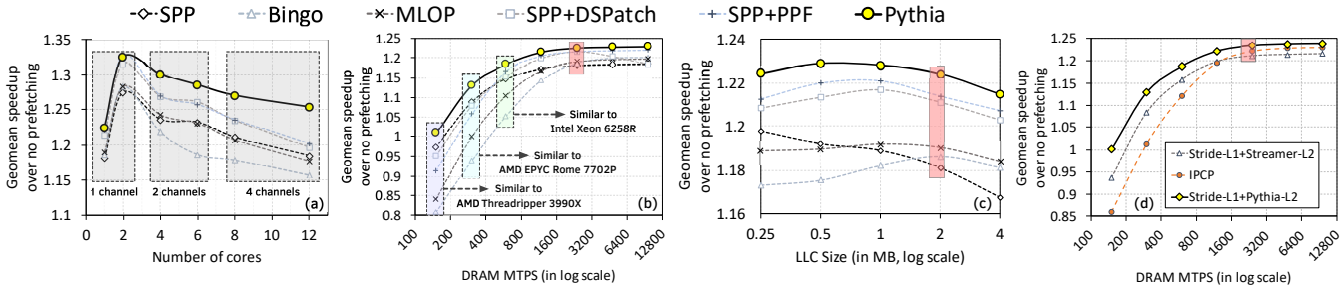

Figure 8: Average performance improvement of prefetchers in systems with varying (a) number of cores, (b) DRAM million transfers per second (MTPS), (c) LLC size, and (d) prefetching level. Each DRAM bandwidth configuration roughly matches MTPS/core of various commercial processors [5, 6, 13]. The baseline bandwidth/LLC configuration is marked in red.

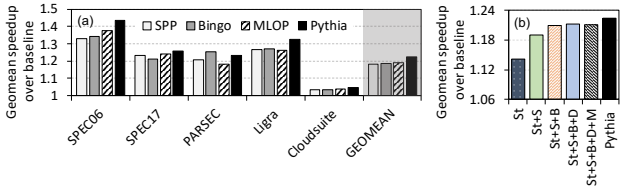

Figure 9: Performance improvement in single-core workloads. St=Stride, S=SPP, B=Bingo, D=DSPatch, and M=MLOP.

in *every* workload category in the four-core system. On average, Pythia outperforms MLOP, Bingo, and SPP by 5.8%, 8.2%, and 6.5% respectively. Second, unlike in the single-core system, Pythia outperforms Bingo in PARSEC by 3.0% in the four-core system. This is due to Pythia's ability to dynamically increase prefetch accuracy during high DRAM bandwidth usage.

Fig. 10(b) shows that Pythia outperforms the combination of stride, SPP, Bingo, DSPatch, and MLOP prefetchers by 4.9% on average. Unlike in the single-core system, combining more prefetchers on top of stride+SPP in four-core system lowers the overall performance gain. This is due to the additive increase in the overpredictions made by each individual prefetcher, which leads to performance degradation in the bandwidth-constrained four-core system. Pythia's RL-based framework holistically learns to prefetch using multiple program features and generates fewer overpredictions, outperforming all combinations of all individual prefetchers.

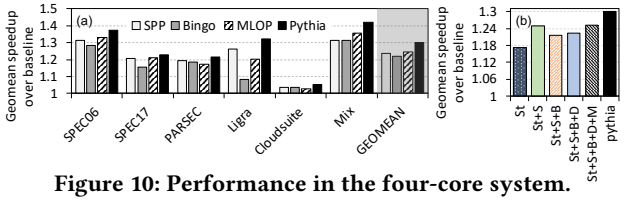

Figure 10: Performance in the four-core system.

6.3.3 ***Benefit of Memory Bandwidth Usage Awareness.*** To demonstrate the benefit of Pythia's awareness of system memory bandwidth usage, we compare the performance of the full-blown Pythia with a new version of Pythia that is oblivious to system memory bandwidth usage. We create this bandwidth-oblivious version of Pythia by setting the high and low bandwidth usage variants of the rewards $\mathcal{R}_{IN}$ and $\mathcal{R}_{NP}$ to the same value (i.e., essentially removing the bandwidth usage distinction from the reward values). More specifically, we set $\mathcal{R}_{IN}^{H} = \mathcal{R}_{IN}^{L} = -8$ and $\mathcal{R}_{NP}^{H} = \mathcal{R}_{NP}^{L} = -4$. Fig. 11 shows the performance benefit of the memory bandwidth-oblivious Pythia normalized to the basic Pythia as we vary the DRAM bandwidth. The key takeaway is that the bandwidth-oblivious Pythia loses performance by up to 4.6% on average across all single-core traces when the available memory bandwidth is low (150-MTPS to 600-MTPS configuration). However, when the available memory bandwidth is high (1200-MTPS to 9600-MTPS), the memory bandwidth-oblivious Pythia provides similar performance improvement to the basic Pythia. We conclude that, memory bandwidth awareness gives Pythia the ability to provide robust performance benefits across a wide range of system configurations.

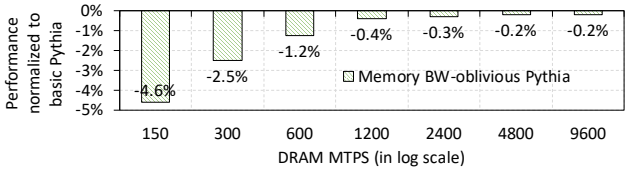

Figure 11: Performance of memory bandwidth-oblivious Pythia versus the basic Pythia.

6.4 Performance on Unseen Traces

To demonstrate Pythia's ability to provide performance gains across workload traces that are not used at all to tune Pythia, we evaluate Pythia using an additional 500 traces from the second value prediction championship [20] on both single-core and four-core systems. These traces are classified into floating-point, integer, crypto, and server categories and each of them has at least 3 LLC MPKI in the baseline without prefetching. No prefetcher, including Pythia, has been tuned on these traces. In the single-core system, Pythia outperforms MLOP, Bingo, and SPP on average by 8.3%, 3.5%, and 4.9%, respectively, across these traces. In the four-core system, Pythia outperforms MLOP, Bingo, and SPP on average by 9.7%, 5.4%, and 6.7%, respectively. We conclude that, Pythia, tuned on a set of workload traces, provides equally high (or even better) performance benefits on unseen traces for which it has not been tuned.

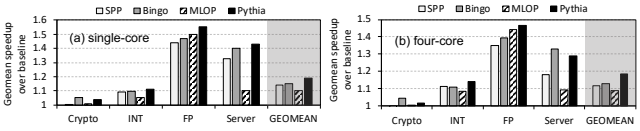

Figure 12: Performance on unseen traces.

6.5 Understanding Pythia Using a Case Study

We delve deeper into an example workload trace, `459.GemsFDTD-1320B`, from SPEC CPU2006 suite to provide more insight into Pythia's prefetching strategy and benefits. In this trace, the top two most selected prefetch offsets by Pythia are +23 and +11, which cumulatively account for nearly 72% of all offset selections. For each of these offsets, we examine the program feature value that selects that offset the most. For simplicity, we only focus on the `PC+Delta` feature here. The `PC+Delta` feature values `0x436a81+0` and `0x4377c5+0` select the offsets +23 and +11 the most, respectively. Fig. 13(a) and (b) show the Q-value curve of different actions for these feature. The x-axis shows the number of Q-value updates to the corresponding feature. Each color-coded line represents the Q-value of the respective action.

As Fig. 13(a) shows, the Q-value of action +23 for feature value `0x436a81+0` consistently stays higher than all other actions (only three other representative actions are shown in 13(a)). This means Pythia actively favors to prefetch using +23 offset whenever the PC `0x436a81` generates the first load to a physical page (hence the delta 0). By dumping the program trace, we indeed find that whenever PC `0x436a81` generates the first load to a physical page, there is only one more address demanded in that page that is 23 cachelines ahead from the first loaded cacheline. In this case, the positive reward for generating a correct prefetch with offset +23 drives the Q-value of +23 much higher than those of other offsets and Pythia successfully uses the offset +23 for prefetch request generation given the feature value `0x436a81+0`. We see similar a trend for the feature value `0x4377c5+0` with offset +11 (Fig. 13(b)).

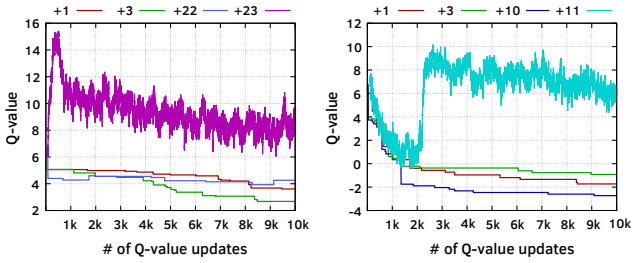

Figure 13: Q-value curves of `PC+Delta` feature values (a) `0x436a81+0` and (b) `0x4377c5+0` in `459.GemsFDTD-1320B`.

6.6 Performance Benefits via Customization

In this section, we show two examples of Pythia's online customization ability to extract even higher performance gain than the baseline Pythia configuration in target workload suites. First, we customize Pythia's reward level values for the `Ligra` graph processing workloads. Second, we customize the program features used by Pythia for the SPEC CPU2006 workloads.

6.6.1 ***Customizing Reward Levels.*** For workloads from the Ligra graph processing suite, we observe a general trend that a prefetcher with higher prefetch accuracy typically provides higher performance benefits. This is because any incorrect prefetch request wastes precious main memory bandwidth, which is already heavily used by the demand requests of the workload. Thus, to improve Pythia's performance benefit in the Ligra suite, we create a new *strict configuration* of Pythia that favors *not to prefetch* over generating inaccurate prefetches. We create this strict configuration by simply reducing the reward level values for inaccurate prefetch (i.e., $\mathcal{R}_{IN}^{H} = -22$ and $\mathcal{R}_{IN}^{L} = -20$) and increasing the reward level values for no prefetch (i.e., $\mathcal{R}_{NP}^{H} = \mathcal{R}_{NP}^{L} = 0$).

Fig. 14 shows the percentage of the total runtime the workload spends in different bandwidth usage buckets in primary y-axis and the overall performance improvement in the secondary y-axis for each competing prefetcher in one example workload from the `Ligra` suite, `Ligra-CC`. We make two key observations. First, with MLOP and Bingo prefetchers enabled, `Ligra-CC` spends a much higher percentage of runtime consuming more than half of the peak DRAM bandwidth than in the no prefetching baseline. As a result, MLOP and Bingo underperforms the no prefetching baseline by 11.8% and 1.8%, respectively. In contrast, basic Pythia leads to only a modest memory bandwidth usage overhead, and outperforms the no prefetching baseline by 6.9%. Second, in the strict configuration, Pythia has even less memory bandwidth usage overhead, and provides 3.5% higher performance than the basic Pythia configuration (10.4% over the no prefetching baseline), without any hardware changes. Fig. 15 shows the performance benefits of the basic and strict Pythia configurations for all workloads from `Ligra`. The key takeaway is that by simply changing the reward level values via configuration registers on the silicon, strict Pythia provides up to 7.8% (2.0% on average) higher performance than basic Pythia. We conclude that the objectives of Pythia can be easily customized via simple configuration registers for target workload suites to extract even higher performance benefits, without any changes to the underlying hardware.

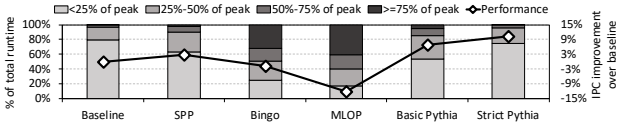

Figure 14: Performance and main memory bandwidth usage of prefetchers in `Ligra-CC`.

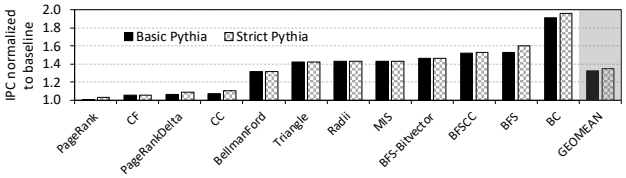

Figure 15: Performance of the basic and strict Pythia configurations on the `Ligra` workload suite.

6.6.2 ***Customizing Feature Selection.*** To maximize the performance benefits of Pythia on the SPEC CPU2006 workload suite, we run all one-combination and two-combination of program features from the initial set of 32 supported features. For each workload, we fine-tune Pythia using the feature combination that provides the highest performance benefit. We call this the *feature-optimized configuration* of Pythia for SPEC CPU2006 suite. Fig. 16 shows the performance benefits of the basic and optimized configurations of Pythia for all SPEC CPU2006 workloads. The key takeaway is that by simply fine-tuning the program feature selection, Pythia delivers up to 5.1% (1.5% on average) performance improvement on top of the basic Pythia configuration.

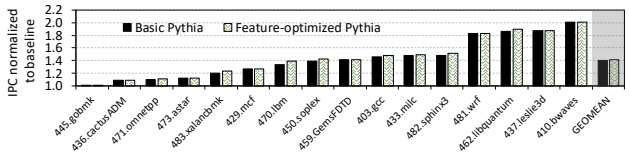

Figure 16: Performance of the basic and feature-optimized Pythia on the SPEC CPU2006 suite.

6.7 Overhead Analysis

To accurately estimate Pythia's chip area and power overheads, we implement the full-blown Pythia, including all fixed-point adders, multipliers, and the pipelined QVStore search operation (§4.2.2), using the Chisel [8] hardware design language (HDL). We extensively verify the functional correctness of the resultant register transfer logic (RTL) design and synthesize the RTL design using Synopsys Design Compiler [23] and 14-nm library from GlobalFoundries [10] to estimate Pythia's area and power overhead. Pythia consumes 0.33 mm2 of area and 55.11 mW of power in each core. The QVStore component consumes 90.4% and 95.6% of the total area and power of Pythia, respectively. With respect to the overall die area and power consumption of a 4-core desktop-class Skylake processor with the lowest TDP budget [11], and a 28-core server-class Skylake processor with the highest TDP budget, Pythia (implemented in all cores) incurs area & power overheads of only 1.03% & 0.4%, and 1.33% & 0.75%, respectively. We conclude that Pythia's performance benefits come at a very modest cost in area and power overheads across a variety of commercial processors.

Table 8: Area and power overhead of Pythia

Pythia's area: 0.33 mm2/core; **Pythia's power**: 55.11 mW/core

Overhead compared to real systems	Area	Power
4-core Skylake D-2123IT, 60W TDP [11]	1.03%	0.37%
18-core Skylake 6150, 165W TDP [12]	1.24%	0.60%
28-core Skylake 8180M, 205W TDP [14]	1.33%	0.75%

7 OTHER RELATED WORKS

To our knowledge, Pythia is the first RL-based customizable prefetching framework that can learn to prefetch using multiple different program features and system-level feedback information inherent to its design, to provide performance benefits across a wide range of workloads and changing system configurations. We already compare Pythia against five state-of-the-art prefetching proposals [27, 30, 32, 78, 111] in §6. In this section, we qualitatively compare Pythia against other prior prefetching techniques.

Traditional Prefetchers. We divide the traditional prefetching algorithms into three broad categories: precomputation-based, temporal, and spatial. Precomputation-based prefetchers (e.g., runahead [46, 59, 60, 63, 96–102] and helper-thread execution [33, 40, 41, 45, 74, 75, 88, 107, 120, 129, 142, 145]) pre-execute program code to generate future memory requests. These prefetchers can generate highly-accurate prefetches even when no recognizable pattern exists in program memory requests. However, precomputation-based prefetchers usually have high design complexity. Pythia is not a precomputation-based proposal. It finds patterns in past memory request addressed to generate prefetch requests.

Temporal prefetchers [26, 29, 36, 37, 42, 52, 62, 66, 72, 77, 121, 130–132, 134, 135] memorize long sequences of cacheline addresses demanded by the processor. When a previously-seen address is encountered again, a temporal prefetcher issues prefetch requests to addresses that previously followed the currently-seen cacheline address. However, temporal prefetchers usually have high storage requirements (often multi-megabytes of metadata storage, which necessitates storing metadata in memory [66, 121, 130]). Pythia requires only 25.5KB of storage, which can easily fit inside a core.

Spatial prefetchers [25, 27, 30, 32, 35, 56, 65, 73, 78–80, 90, 103, 106, 111, 112, 122, 123] predict a cacheline delta or spatial bit-pattern by learning program access patterns over different spatial memory regions. Spatial prefetchers provide high-accuracy prefetches, usually with lower storage overhead than temporal prefetchers. We already compare Pythia with other spatial prefetchers [27, 30, 32, 78, 111] and show higher performance benefits.

Machine Learning (ML) in Computer Architecture. Prior works apply ML techniques in computer architecture in two major ways: (1) to design adaptive, data-driven algorithms, and (2) to explore the large microarchitectural design-space. Researchers have proposed ML-based algorithms for various microarchitectural tasks like memory scheduling [64, 94], cache management [28, 87, 110, 113, 127], branch prediction [57, 67–70, 125, 126, 139, 140, 146], address translation [89] and hardware prefetching [61, 104, 105, 114–116, 141]. Pythia provides three key advantages over prior ML-based prefetchers. First, Pythia can learn to prefetch from multiple program features and system-level feedback information inherent to its design. Second, Pythia can be customized online. Third, Pythia incurs low hardware overhead. Researchers have also explored ML techniques to explore the large microarchitectural design space, e.g., NoC design [39, 43, 44, 50, 54, 84, 128, 136, 137, 143], chip placement optimization [91], hardware resource assignment for accelerators [76]. These works are orthogonal to Pythia.

8 CONCLUSION

We introduce Pythia, the first customizable prefetching framework that formulates prefetching as a reinforcement learning (RL) problem. Pythia autonomously learns to prefetch using multiple program features and system-level feedback information to predict memory accesses. Our extensive evaluations show that Pythia not only outperforms five state-of-the-art prefetchers but also provides robust performance benefits across a wide-range of workloads and system configurations. Pythia's benefits come with very modest area and power overheads. We believe and hope that Pythia would encourage the next generation of data-driven autonomous prefetchers that automatically learn far-sighted prefetching policies by interacting with the system. Such prefetchers can not only improve performance and efficiency under a wide variety of workloads and system configurations, but also reduce the system architect's burden in designing sophisticated prefetching mechanisms.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We thank all SAFARI Research Group members, especially Skanda Koppula, for insightful feedback. We acknowledge the generous gifts provided by our industrial partners: Google, Huawei, Intel, Microsoft, and VMware. This work is also in part supported by SRC research grant as a part of AI Hardware program. The first author thanks his departed father, whom he lost in COVID-19 pandemic.

REFERENCES

[1] 2nd Cache Replacement Championship. https://crc2.ece.tamu.edu.
[2] 2nd Data Prefetching Championship. http://comparch-conf.gatech.edu/dpc2/.
[3] 3rd Data Prefetching Championship. https://dpc3.compas.cs.stonybrook.edu.
[4] 6th Generation Intel® Processor Family. https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-spec-update.html.
[5] AMD Ryzen Threadripper 3990X. https://en.wikichip.org/wiki/amd/ryzen_threadripper/3990x.
[6] AMD Zen2 EPYC 7702P. https://en.wikichip.org/wiki/amd/epyc/7702p.
[7] ChampSim. https://github.com/ChampSim/ChampSim.
[8] Chisel/FIRRTL Hardware Compiler Framework. https://www.chisel-lang.org.
[9] Disclosure of Hardware Prefetcher Control on Some Intel® Processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html.
[10] GlobalFoundries 14nm FinFET Technology. https://www.globalfoundries.com/sites/default/files/product-briefs/pb-14lpp.pdf.
[11] Intel Xeon D-2123IT. https://en.wikichip.org/wiki/intel/xeon_d/d-2123it.
[12] Intel Xeon Gold 6150. https://en.wikichip.org/wiki/intel/xeon_gold/6150.
[13] Intel Xeon Gold 6258R. https://en.wikichip.org/wiki/intel/xeon_gold/6258r.
[14] Intel Xeon Platinum 8180M. https://en.wikichip.org/wiki/intel/xeon_platinum/8180m.
[15] JEDEC-DDR4. https://www.jedec.org/sites/default/files/docs/JESD79-4.pdf.
[16] PARSEC. http://parsec.cs.princeton.edu/.
[17] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[18] Pythia. https://en.wikipedia.org/wiki/Pythia.
[19] Pythia GitHub Repository. https://github.com/CMU-SAFARI/Pythia.
[20] Second Championship Value Prediction (CVP-2). https://www.microarch.org/cvp1/cvp2/rules.html.
[21] SPEC CPU 2006. https://www.spec.org/cpu2006/.
[22] SPEC CPU 2017. https://www.spec.org/cpu2017/.
[23] Synopsys DC Ultra. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.
[24] J. S. Albus. A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*. 1975.
[25] Jean-Loup Baer and Tien-Fu Chen. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *SC*. 1991.
[26] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino Temporal Data Prefetcher. In *HPCA*. 2018.
[27] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo Spatial Data Prefetcher. In *HPCA*. 2019.
[28] Arjun Balasubramanian, Adarsh Kumar, Yuhan Liu, Han Cao, Shivaram Venkataraman, and Aditya Akella. Accelerating Deep Learning Inference via Learned Caches. 2021.
[29] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, et al. Correlated load-address predictors. *ISCA*. 1999.
[30] Rahul Bera, Anant V Nori, Onur Mutlu, and Sreenivas Subramoney. DSPatch: Dual Spatial Pattern Prefetcher. In *MICRO*. 2019.
[31] James Bergstra and Yoshua Bengio. Random Search for Hyper-parameter Optimization. *The Journal of Machine Learning Research* 13, 1. 2012.
[32] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-Based Prefetch Filtering. In *ISCA*. 2019.
[33] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *ISCA*. 1999.
[34] M. J. Charney and T. R. Puzak. Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite. *IBM Journal of Research and Development*. 1997.
[35] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. In *IEEE TC*. 1995.
[36] Trishul M Chilimbi and Martin Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *PLDI*. 2002.
[37] Yuan Chou. Low-cost Epoch-based Correlation Prefetching for Commercial Applications. In *MICRO*. 2007.
[38] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 208–214. 2011.
[39] Mark Clark, Avinash Kodi, Razvan Bunescu, and Ahmed Louri. LEAD: Learning-enabled Energy-aware Dynamic Voltage/Frequency Scaling in NoCs. In *DAC*. 2018.
[40] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic Speculative Precomputation. In *MICRO*. 2001.
[41] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, et al. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*. 2001.
[42] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *ASPLOS*. 2002.
[43] Dominic DiTomaso, Travis Boraten, Avinash Kodi, and Ahmed Louri. Dynamic Error Mitigation in NoCs Using Intelligent Prediction Techniques. In *MICRO*. 2016.
[44] Dominic DiTomaso, Ashif Sikder, Avinash Kodi, and Ahmed Louri. Machine Learning Enabled Power-aware Network-on-chip Design. In *DATE*. 2017.
[45] Michel Dubois and Y Song. Assisted Execution. *University of Southern California CENG Technical Report*. 1998.
[46] James Dundas and Trevor Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. In *ICS*. 1997.
[47] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Prefetch-aware Shared Resource Management for Multi-core Systems. In *ISCA*. 2011.
[48] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO*. 2009.
[49] Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *HPCA*. 2009.
[50] Masoumeh Ebrahimi, Masoud Daneshtalab, Fahimeh Farahnakian, Juha Plosila, Pasi Liljeberg, Maurizio Palesi, et al. HARAQ: Congestion-aware Learning Model for Highly Adaptive Routing Algorithm in On-Chip Networks. In *NOCS*. 2012.
[51] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *ASPLOS*. 2012.
[52] Michael Ferdman and Babak Falsafi. Last-touch Correlated Data Streaming. In *ISPASS*. 2007.
[53] Michael Ferdman, Stephen Somogyi, and Babak Falsafi. Spatial Memory Streaming with Rotated Patterns. In *In 1st JILP Data Prefetching Championship*. 2009.
[54] Quintin Fettes, Mark Clark, Razvan Bunescu, Avinash Karanth, and Ahmed Louri. Dynamic Voltage and Frequency Scaling in NoCs with Supervised and Reinforcement Learning techniques. *IEEE TC*. 2018.
[55] John W. C. Fu and Janak H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *ISCA*. 1991.
[56] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO*. 1992.
[57] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A Jiménez. Bit-level Perceptron Prediction for Indirect Branches. In *ISCA*. 2019.
[58] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of machine learning research* 3, Mar. 2003.
[59] Milad Hashemi, Onur Mutlu, and Yale N Patt. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *MICRO*. 2016.
[60] Milad Hashemi and Yale N Patt. Filtered Runahead Execution with a Runahead Buffer. In *MICRO*. 2015.
[61] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, et al. Learning Memory Access Patterns. In *ICML*. 2018.
[62] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: Tag Correlating Prefetchers. In *HPCA*. 2003.
[63] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G Abraham. Effective stream-based and execution-based data prefetching. In *ICS*. 2004.
[64] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA*. 2008.
[65] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access Map Pattern Matching for Data Cache Prefetch. In *ISC*. 2009.
[66] Akanksha Jain and Calvin Lin. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *MICRO*. 2013.
[67] Daniel A Jiménez. Fast path-based neural branch prediction. In *MICRO*. 2003.
[68] Daniel A Jiménez. Multiperspective Perceptron Predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. 2016.
[69] D. A. Jimenez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *HPCA*. 2001.
[70] Daniel A Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *TOCS*. 2002.
[71] Víctor Jiménez, Francisco J Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, Pradip Bose, Francis P O'Connell, et al. Adaptive Prefetching on POWER7: Improving Performance and Power Consumption. *TOPC*. 2014.
[72] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *ISCA*. 1997.
[73] Norman P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *ISCA*. 1990.
[74] Changhee Jung, Daeseob Lim, Jaejin Lee, and Y. Solihin. Helper thread prefetching for loosely-coupled multiprocessor systems. In *IPDPS*. 2006.
[75] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. B-fetch: Branch Prediction Directed Prefetching for Chip-multiprocessors. In *MICRO*. 2014.

[76] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *MICRO*. 2020.
[77] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *HPCA)*. 2000.
[78] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Reddy, Chris Wilkerson, and Zeshan Chishti. Path Confidence based Lookahead Prefetching. In *MICRO*. 2016.
[79] Sushant Kondguli and Michael Huang. Division of labor: A more effective approach to prefetching. In *ISCA*. 2018.
[80] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *ISCA*. 1998.
[81] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-Aware DRAM Controllers. In *MICRO*. 2008.
[82] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *MICRO*. 2009.
[83] PM Lerman. Fitting Segmented Regression Models by Grid Search. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 29, 1. 1980.
[84] Ting-Ru Lin, Drew Penney, Massoud Pedram, and Lizhong Chen. A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study. In *HPCA*. 2020.
[85] Wei-Fen Lin, S.K. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *HPCA*. 2001.
[86] Wei-Fen Lin, S.K. Reinhardt, D. Burger, and T.R. Puzak. Filtering superfluous prefetches using density vectors. In *ICCD*. 2001.
[87] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *ICML*. 2020.
[88] Chi-Keung Luk. Tolerating Memory Latency Through Software-controlled Pre-execution in Simultaneous Multithreading Processors. In *ISCA*. 2001.
[89] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Virtual address translation via learned page table indexes. In *ML for Systems at NeurIPS*. 2018.
[90] Pierre Michaud. Best-offset Hardware Prefetching. In *HPCA*. 2016.
[91] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, et al. A Graph Placement Methodology for Fast Chip Design. *Nature*. 2021.
[92] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, et al. Human-level control through deep reinforcement learning. *Nature*. 2015.
[93] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. Vol. 821. John Wiley & Sons. 2012.
[94] Janani Mukundan and Jose F Martinez. MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler. In *ISCA*. 2012.
[95] Onur Mutlu, Hyesoon Kim, David N Armstrong, and Yale N Patt. Using the First-Level Caches as Filters to Reduce the Pollution Caused by Speculative Memory References. *IJPP*. 2005.
[96] Onur Mutlu, Hyesoon Kim, and Yale N Patt. Address-value Delta (AVD) prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns. In *MICRO*. 2005.
[97] Onur Mutlu, Hyesoon Kim, and Yale N Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *ISCA*. 2005.
[98] Onur Mutlu, Hyesoon Kim, and Yale N Patt. Efficient Runahead Execution: Power-efficient Memory Latency Tolerance. In *IEEE Micro*. 2006.
[99] Onur Mutlu, Hyesoon Kim, Jared Stark, and Yale N Patt. On reusing the results of pre-executed instructions in a runahead execution processor. *IEEE CAL*. 2005.
[100] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *HPCA*. 2003.
[101] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. In *IEEE Micro*. 2003.
[102] Ajeya Naithani, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. Vector Runahead. In *ISCA*. 2021.
[103] S. Pakalapati and B. Panda. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *ISCA*. 2020.
[104] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic Locality and Context-based Prefetching using Reinforcement Learning. In *ISCA*. 2015.
[105] Leeor Peled, Uri Weiser, and Yoav Etsion. A neural network memory prefetcher using semantic locality. *arXiv preprint arXiv:1804.00478*. 2018.
[106] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, et al. Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers. In *HPCA*. 2014.
[107] T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero. Runahead Threads to Improve SMT Performance. In *HPCA*. 2008.
[108] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK. 1994.
[109] George AF Seber and Alan J Lee. *Linear regression analysis*. Vol. 329. John Wiley & Sons. 2012.
[110] Subhash Sethumurugan, Jieming Yin, and John Sartori. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *HPCA*. 2021.
[111] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Multi-lookahead offset prefetching. 2019.
[112] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently Prefetching Complex Address Patterns. In *MICRO*. 2015.
[113] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying Deep learning to The Cache Replacement Problem. In *MICRO*. 413–425. 2019.
[114] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A Neural Hierarchical Sequence Model for Irregular Data Prefetching. In *ML For Systems Workshop, NeurIPS*. 2019.
[115] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A Hierarchical Neural Model of Data Prefetching. In *ASPLOS*. 2021.
[116] Zhan Shi, Kevin Swersky, Daniel Tarlow, Parthasarathy Ranganathan, and Milad Hashemi. Learning execution through neural code fusion. *arXiv preprint arXiv:1906.07181*. 2019.
[117] Julian Shun and Guy E Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146. 2013.
[118] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016.
[119] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. 2018.
[120] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA*. 2002.
[121] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-Temporal Memory Streaming. In *ISCA*. 2009.
[122] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial Memory Streaming. In *ISCA*. 2006.
[123] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-efficiency of Hardware Prefetchers. In *HPCA*. 2007.
[124] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. In *MIT Press*. 2017.
[125] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *TACO*. 2005.
[126] Stephen J Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, and Hong Wang. Improving branch prediction by modeling global history with convolutional neural networks. *arXiv preprint arXiv:1906.09889*. 2019.
[127] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron Learning for Reuse Prediction. In *MICRO*. 2016.
[128] Scott Van Winkle, Avinash Karanth Kodi, Razvan Bunescu, and Ahmed Louri. Extending the Power-efficiency and Performance of Photonic Interconnects for Heterogeneous Multicores with Machine Learning. In *HPCA*. 2018.
[129] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, et al. Helper Threads via Virtual Multithreading on an Experimental Itanium®2 Processor-based Platform. In *ASPLOS*. 2004.
[130] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical Off-chip Meta-data for Temporal Memory Streaming. In *HPCA*. 2009.
[131] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Making Address-correlated Prefetching Practical. *IEEE Micro*. 2010.
[132] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *ISCA*. 2005.
[133] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *MICRO*. 2011.
[134] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal Prefetching Without the Off-Chip Metadata. In *MICRO*. 2019.
[135] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin. Efficient Metadata Management for Irregular Data Prefetching. In *ISCA*. 2019.
[136] Jieming Yin, Yasuko Eckert, Shuai Che, Mark Oskin, and Gabriel H Loh. Toward More Efficient NoC Arbitration: A Deep Reinforcement Learning Approach. In *AIDArc*. 2018.
[137] Jieming Yin, Subhash Sethumurugan, Yasuko Eckert, Chintan Patel, Alan Smith, Eric Morton, et al. Experiences with ML-Driven Design: A NoC Case Study. In *HPCA*. 2020.
[138] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on job scheduling strategies for parallel*

processing. Springer, 44–60. 2003.
[139] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale Patt. BranchNet : Using Offline Deep Learning To Predict Hard-To-Predict Branches. In *MICRO*. 2020.
[140] Siavash Zangeneh, Stephen Pruett, and Yale Patt. Branch prediction with multilayer neural networks: The value of specialization. *Machine Learning for Computer Architecture and Systems*. 2020.
[141] Yuan Zeng and Xiaochen Guo. Long Short Term Memory based Hardware Prefetcher: A Case Study. In *MEMSYS*. 2017.
[142] Weifeng Zhang, Dean M Tullsen, and Brad Calder. Accelerating and Adapting Precomputation Threads for Effcient Prefetching. In *HPCA*. 2007.
[143] Hao Zheng and Ahmed Louri. An Energy-efficient Network-on-chip Design using Reinforcement Learning. In *DAC*. 2019.
[144] X. Zhuang and H.-H.S. Lee. A Hardware-based Cache Pollution Filtering Mechanism for Aggressive Prefetches. In *ICPP*. 2003.
[145] Craig Zilles and Gurindar Sohi. Execution-based Prediction Using Speculative Slices. In *ISCA*. 2001.
[146] Anastasios Zouzias, Kleovoulos Kalaitzidis, and Boris Grot. Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies. *ArXiv* abs/2106.13429. 2021.

A ARTIFACT APPENDIX

A.1 Abstract

We implement Pythia using ChampSim simulator [7]. In this artifact, we provide the source code of Pythia and necessary instructions to reproduce its key performance results. We identify four key results to demonstrate Pythia's novelty:

- Workload category-wise performance speedup of all competing prefetchers (Fig. 9(a)).
- Workload category-wise coverage and overpredictions of all competing prefetchers (Fig. 7).
- Geomean performance comparison with varying DRAM bandwidth from 150-MTPS to 9600-MTPS (Fig. 8(b)).
- Workload category-wise performance speedup of all competing prefetchers (Fig. 10(a)).

The artifact can be executed in any machine with a general-purpose CPU and 52 GB disk space. However, we strongly recommend running the artifact on a compute cluster with `slurm` [138] support for bulk experimentation.

A.2 Artifact Check-list (Meta-information)

- **Compilation:** G++ v6.3.0 or above.
- **Data set:** Download traces using the supplied script.
- **Run-time environment:** Perl v5.24.1
- **Metrics:** IPC, prefetcher's coverage, and accuracy.
- **Experiments:** Generate experiments using supplied scripts.
- **How much disk space required (approximately)?:** 52GB
- **How much time is needed to prepare workflow (approximately)?:** ~ 2 hours. Mostly depends on the time to download traces.
- **How much time is needed to complete experiments (approximately)?:** 3-4 hours using a compute cluster with 480 cores.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5520125

A.3 Description

A.3.1 How to Access. The source code can be downloaded either from GitHub (https://github.com/CMU-SAFARI/Pythia) or from Zenodo (https://doi.org/10.5281/zenodo.5520125).

A.3.2 Hardware Dependencies. Pythia can be run on any system with a general-purpose CPU and at least 52 GB of free disk space.

A.3.3 Software Dependencies. Pythia requires `GCC v6.3.0` and `Perl v5.24.1`. Optionally, Pythia requires `megatools v1.9.98` to download few traces, and Microsoft Excel (tested on v16.51) to reproduce results as presented in the paper.

A.3.4 Data Sets. The ChampSim traces required to evaluate Pythia can be downloaded using the supplied script. Our implementation of Pythia is fully compatible with prior ChampSim traces that are used in previous cache replacement (CRC-2 [1]), data prefetching (DPC-3 [3]) and value-prediction (CVP-2 [20]) championships. We are also releasing a new set of ChampSim traces extracted from Ligra [117] and PARSEC-2.1 [16] suites.

A.4 Installation

(1) Clone Pythia from GitHub repository:

```
$ git clone https://github.com/CMU-SAFARI/Pythia.git
```

(2) Clone Bloomfilter library inside Pythia home and build:

```
$ cd Pythia/
$ git clone https://github.com/mavam/libbf.git libbf/
$ cd libbf/
$ mkdir build && cd build/ && cmake ../
$ make clean && make
```

(3) Build Pythia for single-core and four-core configurations:

```
$ cd $PYTHIA_HOME
$ ./build_champsim.sh multi multi no 1
$ ./build_champsim.sh multi multi no 4
```

(4) Please make sure to set environment variables as:

```
$ source setvars.sh
```

A.5 Experiment Workflow

This section describes steps to generate, and execute necessary experiments. We recommend the reader to follow the README file to know more about each script used in this section.

A.5.1 Preparing Traces.

(1) Download necessary traces as follows:

```
$ mkdir $PYTHIA_HOME/traces
$ cd $PYTHIA_HOME/scripts
$ perl download_traces.pl -csv artifact_traces.csv
-dir $PYTHIA_HOME/traces/
```

(2) If the traces are downloaded in other path, please update the full path in `MICRO21_1C.tlist` and `MICRO21_4C.tlist` inside `$PYTHIA_HOME/experiments` directory appropriately.

A.5.2 Launching Experiments. The following instructions will launch all experiments required to reproduce key results in a local machine. We **strongly** recommend using a compute cluster with `slurm` support to efficiently launch experiments in bulk. To launch experiments using `slurm`, please provide `-local 0` (tested using `slurm v16.05.9`).

(1) Launch single-core experiments as follows:

```
$ cd $PYTHIA_HOME/experiments
$ perl $PYTHIA_HOME/scripts/create_jobfile.pl -exe
$PYTHIA_HOME/bin/perceptron-multi-multi-no-ship-1core
-tlist MICRO21_1C.tlist -exp MICRO21_1C.exp -local 1 > jobfile.sh
$ cd experiments_1C
$ source ../jobfile.sh
```

(2) Launch four-core experiments as follows:

```
$ cd $PYTHIA_HOME/experiments
$ perl $PYTHIA_HOME/scripts/create_jobfile.pl -exe
$PYTHIA_HOME/bin/perceptron-multi-multi-no-ship-4core
```

```
-tlist MICRO21_4C.tlist -exp MICRO21_4C.exp -local 1 > jobfile.sh
$ cd experiments_4C
$ source ../jobfile.sh
```

(3) Please make sure the paths used in `tlist` and `exp` files are appropriately changed before creating the experiment files.

A.5.3 Rolling-up Statistics. We will use `rollup.pl` script to rollup statistics from outputs of all experiments. To automate the process, we will use the following instructions. This will create three comma-separated-value (CSV) files in experiments directory which will be used for evaluation in appendix A.6.

```
$ cd $PYTHIA_HOME/experiments
$ bash automate_rollup.sh
```

A.6 Evaluation

For single-core baseline configuration experiments, we will evaluate three metrics: performance, coverage, and overprediction of each prefetcher. For single-core experiments varying DRAM-bandwidth and four-core experiments, we will evaluate only performance. The performance, coverage and overprediction of a prefetcher X is measured by following equations:

$$Perf_X = \frac{IPC_X}{IPC_{nopref}}$$

$$Coverage_X = \frac{LLC\_load\_miss_{nopref} - LLC\_load\_miss_X}{LLC\_load\_miss_{nopref}}$$

$$Overprediction_X = \frac{LLC\_read\_miss_X - LLC\_read\_miss_{nopref}}{LLC\_read\_miss_{nopref}}$$

To easily calculate the metrics, we are providing a Microsoft Excel template to post-process the rolled-up CSV files. The template has four sheets, three of which bear the same name of the rolled up CSV files. Each sheet is already populated with our collected results, necessary formulas, pivot tables, and charts to reproduce the results presented in the paper. Please follow the instructions to reproduce the results from your own CSV statistics files:

(1) Copy and paste each CSV file in the corresponding sheet's top left corner (i.e., cell A1).
(2) Immediately after pasting, convert the comma-separated rows into columns by going to Data -> Text-to-Coloumns -> Selecting comma as a delimiter. This will replace the already existing data in the sheet with the newly collected data.
(3) *Refresh each pivot table in each sheet* by clicking on them and then clicking Pivot-Table-Analyse -> Refresh.

The reader can also use any other data processor (e.g., Python pandas) to reproduce the same result.

A.7 Expected Results

- In single-core experiments, Pythia should achieve 22% performance improvement over the no prefetching baseline, with 71% prefetch coverage and 27% overpredictions. The next best prefetcher MLOP should achieve 19% performance improvement, with 64% coverage and 110% overpredictions.
- In single-core experiments with DRAM bandwidth scaling, Pythia should achieve the following:
  - In 150-MTPS configuration, Pythia should achieve 0.89% performance improvement over no prefetching baseline, whereas MLOP should *underperform* baseline by 16%.
  - In 9600-MTPS configuration, Pythia should achieve 22.7% performance improvement over no prefetching baseline, whereas MLOP should achieve 19.5%.
- In four-core experiments, Pythia should achieve 30% performance improvement over no prefetching baseline, whereas MLOP should achieve 24%.

A.8 Experiment Customization

- The configuration of every prefetcher can be customized by changing the `ini` files inside the `config` directory.
- The `exp` files can be customized to run new experiments with different prefetcher combinations. More experiment files can be found inside `experiments/extra` directory. One can use the same instructions mentioned in appendix A.5.2 to launch experiments.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

B EXTENDED RESULTS

B.1 Detailed Performance Analysis

B.1.1 ***Single-core.*** Fig. 17 shows the performance line graph of all prefetchers for the 150 single-core workload traces. The workload traces are sorted in ascending order of performance improvement of Pythia over the baseline without prefetching. We make three key observations. First, Pythia outperforms the no-prefetching baseline in every single-core trace, except `623.xalancbmk-592B` (where it underperforms the baseline by 2.1%). `603.bwaves-2931B` enjoys the highest performance improvement of 2.2× over the baseline. Performance of the top 80% of traces improve by at least 4.2% over the baseline. Second, Pythia underperforms Bingo in workloads like `libquantum` due to the heavy streaming nature of memory accesses. As `libquantum` streams through all physical pages, Bingo simply prefetches all cachelines of a page at once just by seeing the first access to the page. As a result Bingo achieves higher timeliness and higher performance than Pythia. Third, Pythia significantly outperforms every competing prefetcher in workloads with irregular access patterns (e.g., `mcf`, `pagerank`). We conclude that Pythia provides consistent performance gains over the no-prefetching baseline and multiple prior state-of-the-art prefetchers over a wide range of workloads. We share a table depicting the single-core performance of every competing prefetcher considered in this paper over the no-prefetching baseline in our GitHub repository: https://github.com/CMU-SAFARI/Pythia.

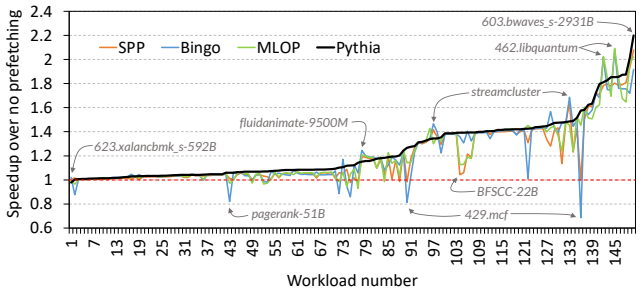

Figure 17: Performance line graph of 150 single-core traces.

B.1.2 ***Four-core.*** Fig. 18 shows the performance line graph of all prefetchers for 272 four-core workload trace mixes (including both homogeneous and heterogeneous mixes). The workload mixes are sorted in ascending order of performance improvement of Pythia over the baseline without prefetching. We make two key observations. First, Pythia outperforms the

baseline without prefetching in all but one four-core trace mix. Pythia provides the highest performance gain in `437.leslie3d-271B` (2.1×) and lowest performance gain in `429.mcf-184B` (-3.5%) over the no-prefetching baseline. Second, Pythia also outperforms (or matches performance) all competing prefetchers in majority of trace mixes. Pythia underperforms Bingo in the `462.libquantum` homogeneous trace mix due to the very regular streaming access pattern. On the other hand, Pythia significantly outperforms Bingo in `Ligra` workloads (e.g., `pagerank`) due to its adaptive prefetching strategy to trade-off coverage for accuracy in high memory bandwidth usage. We conclude that Pythia provides a consistent performance gain over multiple prior state-of-the-art prefetchers over a wide range of workloads even in bandwidth-constrained multi-core systems.

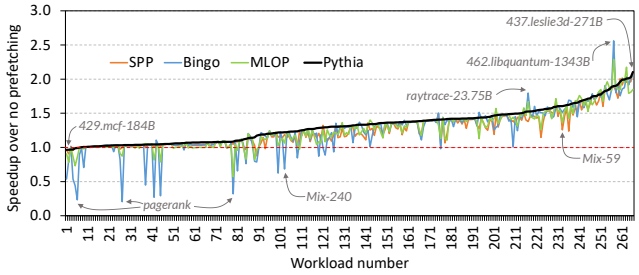

Figure 18: Performance line graph of 272 four-core trace mixes.

B.2 Performance with Different Features

Fig. 19 shows the performance, coverage, and overprediction of Pythia averaged across all single-core traces with different feature combinations during automated feature selection (§4.3.1). For brevity, we show results for all experiments with any-one and any-two combinations of 20 features taken from the full list of 32 features. Both graphs are sorted in ascending order of performance improvement of Pythia over the baseline without prefetching. We make three key observations. First, Pythia's performance gain over the no-prefetching baseline improves from 20.7% to 22.4% by varying the feature combination. We select the feature combination that provides the highest performance gain as the basic Pythia configuration (Table 2). Second, Pythia's coverage and overprediction also change significantly with varying feature combination. Pythia's coverage improves from 66.2% to 71.5%, whereas overprediction improves from 32.2% to 26.7% by changing feature combination. Third, Pythia's performance gain positively correlates with Pythia's coverage in single-core configuration. We conclude that automatic design-space exploration can significantly optimize Pythia's performance, coverage, and overpredictions.

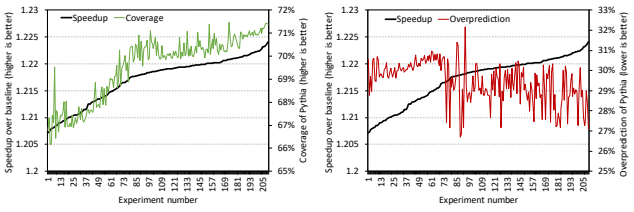

Figure 19: Performance, coverage, and overprediction of Pythia with different feature combinations. The x-axis shows experiments with different feature combinations.

B.3 Performance Sensitivity to Hyperparameters

Fig. 20(a) shows Pythia's performance sensitivity to the exploration rate (ϵ) averaged across all single-core traces. The key takeaway from Figure 20(a) is that Pythia's performance improvement drops sharply if the underlying RL-agent heavily *explores* the state-action space as opposed to exploiting the learned policy. Changing the ϵ-value from 0.002 to 1.0 reduces Pythia's performance improvement by 16.0%. Fig. 20(b) shows Pythia's performance sensitivity to learning rate parameter (α), averaged across all single-core traces. The key takeaway from Fig. 20(b) is that Pythia's performance improvement reduces for both increasing or decreasing the learning rate parameter. Increasing the learning rate reduces the hysteresis in Q-values (i.e., Q-values change significantly with the immediate reward received by Pythia), which reduces Pythia's performance improvement. Similarly, decreasing the learning rate also reduces Pythia's performance as it increases the hysteresis in Q-values. Pythia achieves optimal performance improvement for $\alpha = 0.0065$.

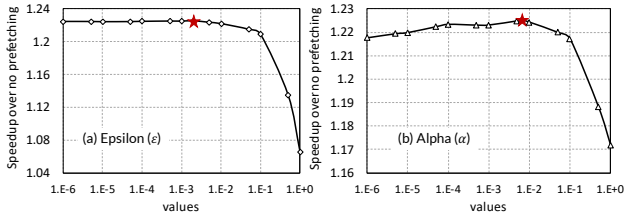

Figure 20: Performance sensitivity of Pythia towards (a) the exploration rate (ϵ), and (b) the learning rate (α) hyperparameter values. The values in basic Pythia configuration are marked in red.

B.4 Comparison to the Context Prefetcher

As we discuss in Section 4.5, unlike Pythia, the context prefetcher (CP [104]) relies on both hardware and software contexts. A tailor-made compiler needs to encode the software contexts using special NOP instructions, which are decoded by the core front-end to pass the context to the CP. For a fair comparison, we implement the context prefetcher using hardware contexts (CP-HW) and show the performance comparison of Pythia and CP-HW in Figure 21. The key takeaway is that Pythia outperforms the CP-HW prefetcher by 5.3% and 7.6% in single-core and four-core configurations, respectively. Pythia's performance improvement over CP-HW mainly comes from two key aspects: (1) Pythia's ability to take memory bandwidth usage into consideration while taking prefetch actions, and (2) the far-sighted predictions made by Pythia as opposed to myopic predictions by CP-HW.

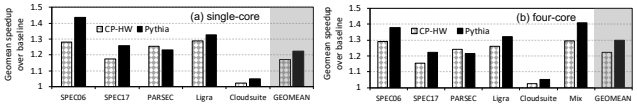

Figure 21: Performance of Pythia vs. the context prefetcher [104] using hardware contexts.

B.5 Comparison to the IBM POWER7 Adaptive Prefetcher

Fig. 22 compares Pythia against the IBM POWER7 adaptive prefetcher [71]. The POWER7 prefetcher dynamically tunes its prefetch aggressiveness (e.g.,

selecting prefetch depth, enabling stride-based prefetching) by monitoring program performance. We make two observations from Fig. 22. First, Pythia outperforms the POWER7 prefetcher by 4.5% in single-core system. This is mostly due to Pythia's ability to capture different types of address patterns than just streaming/stride patterns. Second, Pythia outperforms POWER7 prefetcher by 6.5% in four-core and 6.1% in eight-core systems (not plotted), respectively. The increase in performance improvement from single to four (or eight) core configuration suggests that Pythia is more adaptive than the POWER7 prefetcher.

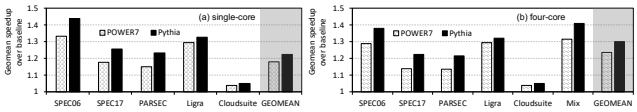

Figure 22: Performance comparison against IBM POWER7 prefetcher [71].