Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera\textsuperscript{1} Konstantinos Kanellopoulos\textsuperscript{1} Anant V. Nori\textsuperscript{2} Taha Shahroodi\textsuperscript{3,1} Sreenivas Subramoney\textsuperscript{2} Onur Mutlu\textsuperscript{1}

\textsuperscript{1}ETH Zürich \textsuperscript{2}Processor Architecture Research Labs, Intel Labs \textsuperscript{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, whichformulates 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 \url{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. \url{https://doi.org/10.1145/3466752.3480114}

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MICRO ’21, October 18–22, 2021, Virtual Event, Greece © 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8557-2/21/10 $15.00

https://doi.org/10.1145/3466752.3480114
(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 prefetcher 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., prefetches 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 prefetcher 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., prefetches of mainly one program feature) for 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.](image)

**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) holistically learns 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, 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 maximize 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

---

1Pythia, 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], Ligo [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 0.37% 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 Ligo 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 with 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 \( S_{t+1} \), 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.](image)

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 using the SARSA [108, 124] algorithm, as shown in Eqn. (1):

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

\( \alpha \) is the learning rate parameter that controls the convergence rate of Q-values. \( \gamma \) 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. \( A \) \( \gamma \) 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 \( \epsilon \)-greedy agent stochastically takes a random action with a low probability of \( \epsilon \) (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.](image-url)
the demanded cacheline address) from a set of candidate prefetch offsets. As every post-L1-cache prefector 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 prefector’s objective. We define five different reward levels as follows.

- **Accurate and timely** ($R_{AT}$). This reward is assigned to an action whose corresponding prefetch address gets demanded after the prefetch fill.
- **Accurate but late** ($R_{AL}$). This reward is assigned to an action whose corresponding prefetch address gets demanded before the prefetch fill.
- **Loss of coverage** ($R_{CL}$). 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** ($R_{IN}$). 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 ($R_{IN}^L$) and inaccurate given high bandwidth usage ($R_{IN}^H$).
- **No-prefetch** ($R_{NP}$). 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 ($R_{NP}^L$) and no-prefetch given high bandwidth usage ($R_{NP}^H$).

By increasing (decreasing) a reward level value, we reinforce (deter) Pythia to collect such rewards from the environment in the future. $R_{AT}$ and $R_{AL}$ are used to guide Pythia to generate more accurate and timely prefetch requests. $R_{CL}$ is used to guide Pythia to generate prefetches within the physical page of the triggering demand request. $R_{IN}$ and $R_{NP}$ 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 (\(\text{1}\)). 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 $R_{AT}$ or $R_{AL}$) 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.) (\(\text{2}\)) and looks up QVStore to find the action with the maximum Q-value for the given state-vector (\(\text{3}\)). Pythia selects the action with the maximum Q-value to generate prefetch request and issues the request to the memory hierarchy (\(\text{4}\)). At the same time, Pythia inserts the selected prefetch action, its corresponding prefetched memory address, and the state-vector into EQ (\(\text{5}\)). 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 (\(\text{6}\)). 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 (\(\text{7}\)). Pythia uses this filled bit in \(\text{1}\) to classify actions that generated timely or late prefetches.\(^3\)

\(\text{1}\)\(\text{2}\)\(\text{3}\)\(\text{4}\)\(\text{5}\)\(\text{6}\)\(\text{7}\)

**Table 1: Example program features**

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

\(^2\)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 prefected 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 ($t_{demand}$).
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 (§4.2.2), and how Pythia assigns rewards to each taken action and how it updates Q-values (§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, \(\phi_1, \phi_2, \ldots, \phi_k\)) 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, \(Q(\phi_i, A)\). 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 \(Q(\phi_i, A)\). 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.

\[
Q(S, A) = \max_{i \in \{1, k\}} Q(\phi_i, A) \tag{3}
\]

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. As Fig. 5(c) shows, to retrieve a feature-action Q-value \( Q(\phi_k, A) \), 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 (3 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. 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 \( R_{SL} \) 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 \( R_{AT} \) or \( R_{AL} \) based on the filled bit of the EQ entry. If

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).
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 $R_{AT}$. Otherwise, Pythia assigns the reward $R_{AL}$. 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 $R^H_{IN}$ or $R^L_{IN}$ 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 configuration1 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,-3,1,0,1,3,5,10,11,12,16,22,23,30,32] |
| Reward Level Values | $R_{AT}=20$, $R_{AL}=12$, $R_{CL}=12$, $R^H_{IN}=-14$, $R^L_{IN}=-8$, $R^H_{NP}=-2$, $R^L_{NP}=-4$ |
| 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-vector across all single-core workloads ($\S$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 | (1) Load cacheline address |
| (2) PC-path (XOR-ed last-3 PCs) | (2) Page number |
| (3) PC XOR-ed branch-PC | (3) Page offset |
| (4) None | (4) Load address delta |
| (5) Sequence of last-4 offsets | (5) Sequence of last-4 offsets |
| (6) Sequence of last-4 deltas | (6) Sequence of last-4 deltas |
| (7) Offset XOR-ed with delta | (7) Offset XOR-ed with delta |
| (8) None | (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 $\S$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., $R_{AT}$, $R_{AL}$, $R_{CL}$, $R^H_{IN}$, $R^L_{IN}$, $R^H_{NP}$ and $R^L_{NP}$) and three hyperparameters (i.e., learning rate $\alpha$, discount factor $\gamma$, and exploration rate $\epsilon$) in three steps. First, we create a test trace suite by randomly selecting 10 workload traces from all of our 150 workload traces ($\S$5). Second, we create a list of tuning configurations using the uniform grid search technique [31, 83]. To do so, we first define a value range for each parameter to be tuned and divide the value range into uniform grids. For example, each of the three hyperparameters ($\alpha$, $\gamma$, and $\epsilon$) can take a value in the range of $[0, 1]$. We divide each hyperparameter range into ten exponentially-sized grids (i.e., $10^0$, $10^{-1}$, $10^{-2}$, etc.) to obtain $10 \times 10 = 1000$ 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 25.5KB 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,
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 |

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 (1C), 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 | Description | Size |
|------|-------------|------|
| 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, SHP [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 CPU2000 [21], SPEC CPU2017 [22], PARSEC 2.1 [16], Ligra [117], and Cloudsuite [51] benchmark suites. For SPEC CPU2006 and SPEC CPU2007 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 100 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, mf, cactusADM, bim, ... |
| SPEC17 | 12 | 18 | gcc, mf, pop2, fotonikld, ... |
| PARSEC | 5 | 11 | canneal, facsim, raytrace, ... |
| Ligra | 13 | 40 | BFS, PageRank, Bellman-ford, ... |
| Cloudsuite | 4 | 53 | cassandra, cloud9, match, ... |

5.2 Prefetchers

We compare Pythia against five state-of-the-art prior prefetchers: SPP [78], SPP+PFP [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 [56, 56, 73] at the L1 level, along with Pythia at the L2 level for such multi-level comparisons.
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.

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, and SPP, and an aggressive SPP with per-socket filtering (PFF [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+PFF 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 1× 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+PFF by 16.9%, 20.2%, 3.7%, and 9.5% respectively in the most bandwidth-constrained configuration with 150 million transfers per second (MTS). In the 9600-MTPS configuration, every prefetcher enjoys ample DRAM bandwidth. Pythia still outperforms MLOP, Bingo, SPP, and SPP+PFF 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 $\frac{1}{4}$× 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+PFF 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 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

### Table 7: Configuration of evaluated prefetchers

| Prefetcher | Configuration | Size   |
|------------|---------------|--------|
| 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 |
| PFF [32]  | Same configuration as in [32] | 39.3 KB |
| Pythia     | 2 features, 2 vaults, 3 planes, 16 actions | 25.5 KB |

| Workload   | SPEC06 | SPEC17 | PARSEC | Ligra | Cloudsuite | AVG |
|------------|--------|--------|--------|-------|------------|-----|
| Coverage   | 9600%  | 9600%  | 9600%  | 9600% | 9600%      |     |
| Uncovered  | 9600%  | 9600%  | 9600%  | 9600% | 9600%      |     |
| Overpredicted | 9600% | 9600% | 9600% | 9600% | 9600% |     |
| AVG        | 9600%  | 9600%  | 9600%  | 9600% | 9600%      |     |

Figure 7: Coverage and overprediction with respect to the baseline LLC misses in the single-core system.
specifically, we set the bandwidth usage distinction from the reward values. More precisely, we create this bandwidth-oblivious version of Pythia by setting the high and low bandwidth usage variants of the rewards $\mathcal{R}_H$ and $\mathcal{R}_L$ to the same value (i.e., essentially removing the bandwidth usage distinction from the reward values). More specifically, we set $\mathcal{R}_H = \mathcal{R}_L = 8$ and $\mathcal{R}_N^H = \mathcal{R}_N^L = -4$. 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.

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.
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 prefetching 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)).

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., 0 \&^{H}_{IN} = -22 \text{ and } 0 \&^{L}_{IN} = -20) and increasing the reward level values for no prefetch (i.e., 0 \&^{H}_{NP} = 0 \text{ and } 0 \&^{L}_{NP} = 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 13: Q-value curves of PC+Delta feature values (a) 0x436a81+0 and (b) 0x4377c5+0 in 459.GemsFDTD-1320B.

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.

---

R. Bera and K. Kanellopoulos, et al.
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. 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 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 supported in part by the Semiconductor Research Corporation and the ETH Future Computing Laboratory. The first author thanks his departed father, whom he lost in COVID-19 pandemic.
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: C++ 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.4.1 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.4.2 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:

```
$ cd PYTHON_HOME/traces/
$ perl download_traces.pl -csv artifact_traces.csv
```

(2) If the traces are downloaded in other path, please update the full path in MICRO21_1C.tlist and MICRO21_4C.tlist inside PYTHON_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 PYTHON_HOME/experiments
$ perl PYTHON_HOME/scripts/create_jobfile.pl -exe PYTHON_HOME/bin/perceptron-multi-multi-no-ship-1core -tlist MICRO21_1C.tlist -exp MICRO21_1C.exp -local 0 > jobfile.sh
$ cd experiments_1C
$ source jobfile.sh
```

(2) Launch four-core experiments as follows:

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

A.5.3 Examples of experiments launched.

```
$ cd PYTHON_HOME/experiments
$ perl PYTHON_HOME/bin/megatools.py PERC no prof -tlist MICRO21_1C.tlist -exp MICRO21_1C.exp -local 0 > jobfile.sh
```

A.5.4 Notes. We recommend the reader to follow the README file to know more about each script used in this section.

A.6.1 Preparing Experiments.

(1) Generate experiments using supplied scripts.

```
$ perl create_jobfile.pl
```

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

A.6.2 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 PYTHON_HOME/experiments
$ perl PYTHON_HOME/bin/megatools.py PERC no prof
```

(2) Launch four-core experiments as follows:

```
$ cd PYTHON_HOME/experiments
$ perl PYTHON_HOME/bin/megatools.py PERC no prof
```
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

- **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 2.2x 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 1bquantum due to the heavy streaming nature of memory accesses. As 1bquantum 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-SAFA/Pythia.

![Figure 17: Performance line graph of 150 single-core traces.](image-url)

- **Four-core.** Fig. 18 shows the performance line graph of all prefetchers for 274 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. 1es11e3d+271B (2.1x) and lowest performance gain in 429. mcf+1848 (+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 462.1i*quantum 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.

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

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 ($\epsilon$) 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 $\epsilon$-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 ($\alpha$), 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 ($\epsilon$), and (b) the learning rate ($\alpha$) 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].

B.6 Performance Sensitivity to Number of Warmup Instructions
Fig. 23 shows performance sensitivity of all prefetchers to the number of warmup instructions averaged across all single-core traces. Our baseline simulation configuration uses 100 million warmup instructions. The key takeaway from Fig. 23 is that Pythia consistently outperforms prior prefetchers in a wide range of simulation configurations using different number of warmup instructions. In the baseline simulation configuration using 100M warmup instructions, Pythia outperforms MLOP, Bingo, and SPP by 3.4%, 3.8%, and 4.4% respectively. In a simulation configuration with no warmup instruction, Pythia continues to outperform MLOP, Bingo, and SPP by 2.8%, 3.7%, and 4.2% respectively. We conclude that, Pythia can quickly learn to prefetch from a program’s memory access pattern and provides higher performance than other heuristics-based prefetching techniques over a wide range of simulation configurations using different number of warmup instructions.

Figure 23: Performance sensitivity of all prefetchers to number of warmup instructions.