Pythia
A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, Onur Mutlu

https://github.com/CMU-SAFARI/Pythia
Executive Summary

• **Background:** Prefetchers predict addresses of future memory requests by associating memory access patterns with program context (called feature)

• **Problem:** Three key shortcomings of prior prefetchers:
  - Predict mainly using a **single program feature**
  - Lack **inherent system awareness** (e.g., memory bandwidth usage)
  - Lack **in-silicon customizability**

• **Goal:** Design a prefetching framework that:
  - Learns from **multiple features** and **inherent system-level feedback**
  - Can be **customized in silicon** to use different features and/or prefetching objectives

• **Contribution:** Pythia, which formulates prefetching as reinforcement learning problem
  - Takes **adaptive** prefetch decisions using multiple features and system-level feedback
  - Can be **customized in silicon** for target workloads via simple configuration registers
  - Proposes a **realistic and practical** implementation of RL algorithm in hardware

• **Key Results:**
  - Evaluated using a wide range of workloads from SPEC CPU, PARSEC, Ligra, Cloudsuite
  - Outperforms best prefetcher (in 1-core config.) by **3.4%, 7.7% and 17%** in 1/4/bw-constrained cores
  - Up to **7.8% more performance** over basic Pythia across Ligra workloads via simple customization

**SAFARI**

[https://github.com/CMU-SAFARI/Pythia](https://github.com/CMU-SAFARI/Pythia)
Talk Outline

Key Shortcomings of Prior Prefetchers

Formulating Prefetching as Reinforcement Learning

Pythia: Overview

Evaluation of Pythia and Key Results

Conclusion
Prefetching Basics

• Predicts addresses of **long-latency memory requests** and fetches data before the program demands it

• Associates access patterns from past memory requests with program context information

  **Program Feature** → Access Pattern

• **Example program features**
  - Program counter (PC)
  - Page number
  - Page offset
  - Cacheline delta
  - ...
  - Or a combination of these attributes
Key Shortcomings in Prior Prefetchers

- We observe three key shortcomings that significantly limit performance benefits of prior prefetchers.

1. Predict mainly using a single program feature

2. Lack inherent system awareness

3. Lack in-silicon customizability

SAFARI
(1) Single-Feature Prefetch Prediction

- Provides **good** performance gains mainly on workloads where the **feature-to-pattern correlation** exists.

![Graph showing IPC improvement over baseline (%)](image)

- **Bingo** [1] performs better than **SPP** [2].

---

[1] Bakshaliopour et al., HPCA’19  
[2] Kim et al., MICRO’16
(1) Single-Feature Prefetch Prediction

- Provides **good performance gains** mainly on workloads where the **feature-to-pattern correlation** exists

Relying on a **single feature** for prediction leaves significant performance improvement on table

<table>
<thead>
<tr>
<th>IPC improvement (%)</th>
<th>482.sphinx3-417B</th>
<th>PARSEC-Canneal</th>
<th>PARSEC-Facesim</th>
<th>459.GemsFDTD-765B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bingo [1]</td>
<td>15.4%</td>
<td>3.5%</td>
<td>5.5%</td>
<td>SPP [2] performs better</td>
</tr>
<tr>
<td>SPP [2]</td>
<td>4.6%</td>
<td>10%</td>
<td>15%</td>
<td>Bingo [1] performs better</td>
</tr>
</tbody>
</table>

(2) Lack of Inherent System Awareness

- Little understanding of **undesirable effects** (e.g., memory bandwidth usage, cache pollution, ...)
  - Performance loss in **resource-constrained** configurations

```
<table>
<thead>
<tr>
<th></th>
<th>Ligra-CC</th>
<th>PARSEC-Canneal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction of LLC misses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPP</td>
<td>0%</td>
<td>-2%</td>
</tr>
<tr>
<td>Bingo</td>
<td>0%</td>
<td>-2%</td>
</tr>
<tr>
<td>Pythia</td>
<td>2%</td>
<td>4%</td>
</tr>
</tbody>
</table>

Similar coverage

Lower overpredictions

Yet, **lower** performance
```
(2) Lack of Inherent System Awareness

• Little understanding of undesirable effects (e.g., memory bandwidth usage, cache pollution, ...)
  - Performance loss in resource-constrained configurations

Prefetchers often lose performance due to lack of inherent system awareness.

Similar coverage  Lower overpredictions  Yet, lower performance
(3) Lack of In-silicon Customizability

- Feature **statically** selected at design time
  - **Rigid hardware** designed specifically to exploit that feature

- **No way to change** program feature and/or change prefetcher’s objective **in silicon**
  - **Cannot adapt** to a wide range of workload demands

*Design from scratch ➔ Verify ➔ Fabricate*
Our Goal

A **prefetching framework** that can:

1. Learn to prefetch using **multiple features** and **inherent system-level feedback** information

2. Be **easily customized in silicon** to use different features and/or change prefetcher’s objectives
Our Proposal

Pythia

Formulates prefetching as a reinforcement learning problem

Pythia is named after the oracle of Delphi, who is known for her accurate prophecies
https://en.wikipedia.org/wiki/Pythia
Talk Outline

- Key Shortcomings of Prior Prefetchers
- Formulating Prefetching as Reinforcement Learning
- Pythia: Overview
- Evaluation of Pythia and Key Results
- Conclusion
Basics of Reinforcement Learning (RL)

• Algorithmic approach to learn to take an action in a given situation to maximize a numerical reward

Agent

Environment

• Agent stores Q-values for every state-action pair
  - Expected return for taking an action in a state
  - Given a state, selects action that provides highest Q-value
Formulating Prefetching as RL

Agent

State ($S_t$)  Reward ($R_{t+1}$)  Action ($A_t$)

Environment

Prefetcher

Features of memory request to address $A$ (e.g., PC)

Processor & Memory Subsystem

Reward

Prefetch from address $A$ + offset ($O$)
What is State?

• **k-dimensional** vector of features
  \[ S \equiv \{\phi_1^S, \phi_2^S, \ldots, \phi_k^S\} \]

• Feature = control-flow + data-flow

• **Control-flow examples**
  - PC
  - Branch PC
  - Last-3 PCs, ...

• **Data-flow examples**
  - Cacheline address
  - Physical page number
  - Delta between two cacheline addresses
  - Last 4 deltas, ...
What is State?

Example of a state information

$S = \{ \text{PC+Delta, Sequence of last-4 deltas} \}$

- Feature-1 ($\phi_1$)
  - PC (Control-flow info.)
  - Cacheline Delta (Data-flow info.)

- Feature-2 ($\phi_2$)
  - Seq. of last-4 deltas (Data-flow info.)
What is Action?

Given a demand access to address A the action is to **select prefetch offset “O”**

- **Action-space**: 127 actions in the range [-63, +63]
  - For a machine with 4KB page and 64B cacheline

- Upper and lower limits ensure prefetches do not cross physical page boundary

- A **zero offset** means **no prefetch** is generated

- We further **prune** action-space by design-space exploration
What is Reward?

• Defines the **objective** of Pythia

• Encapsulates two metrics:
  - **Prefetch usefulness** (e.g., accurate, late, out-of-page, ...)
  - **System-level feedback** (e.g., mem. b/w usage, cache pollution, energy, ...)

• We demonstrate Pythia with **memory bandwidth usage** as the system-level feedback in the paper
What is Reward?

- **Seven** distinct reward levels
  - Accurate and timely ($R_{AT}$)
  - Accurate but late ($R_{AL}$)
  - Loss of coverage ($R_{CL}$)
  - Inaccurate
    - With low memory b/w usage ($R_{IN-L}$)
    - With high memory b/w usage ($R_{IN-H}$)
  - No-prefetch
    - With low memory b/w usage ($R_{NP-L}$)
    - With high memory b/w usage ($R_{NP-H}$)

- Values are set at design time via **automatic design-space exploration**
  - Can be **customized** further in silicon for higher performance
Steering Pythia’s Objective via Reward Values

• Example reward configuration for
  - Generating **accurate prefetches**
  - Making **bandwidth-aware** prefetch decisions

1. Highly prefers to generate accurate prefetches

2. Prefers not to prefetch if memory bandwidth usage is low

3. **Strongly** prefers not to prefetch if memory bandwidth usage is high

**RAT** = Accurate & timely; **AL** = Accurate & late; **NP** = No-prefetching; **IN** = Inaccurate; **H** = High mem. b/w; **L** = Low mem. b/w
Steering Pythia’s Objective via Reward Values

• Customizing reward values to make Pythia **conservative** towards prefetching

\[ R_{\text{NP-L}} \quad R_{\text{NP-H}} \]

\[ R_{\text{AL}} \quad R_{\text{AT}} \]

\[ R_{\text{IN-H}} \quad R_{\text{IN-L}} \]

AT = Accurate & timely; AL = Accurate & late; NP = No-prefetching; IN = Inaccurate;
H = High mem. b/w; L = Low mem. b/w

**1**
Highly prefers to generate accurate prefetches

**2**
Otherwise prefers not to prefetch
Steering Pythia’s Objective via Reward Values

- Customizing reward values to make Pythia conservative towards prefetching

**Strict Pythia configuration**

1. Server-class processors
2. Bandwidth-sensitive workloads

\[\text{AT} = \text{Accurate & timely}; \quad \text{AL} = \text{Accurate & late}; \quad \text{NP} = \text{No-prefetching}; \quad \text{IN} = \text{Inaccurate}; \quad \text{H} = \text{High mem. b/w}; \quad \text{L} = \text{Low mem. b/w}\]
**Pythia Overview**

- **Q-Value Store**: Records Q-values for *all* state-action pairs
- **Evaluation Queue**: A FIFO queue of recently-taken actions

**Diagram Description**

1. **Demand Request**
   - Assign reward to corresponding EQ entry

2. **State Vector**
   - Look up QVStore

3. **Q-Value Store (QVStore)**
   - Find the Action with max Q-Value

4. **Generate prefetch**

5. **Insert prefetch action & State-Action pair in EQ**

6. **Evict EQ entry and update QVStore**

7. **Set filled bit**

**Memory Hierarchy**

**Equations and Process**

- Assign reward to the corresponding EQ entry.
- Look up QVStore to find the action with the maximum Q-Value.
- Generate a prefetch based on the max Q-Value.
- Insert the prefetch action along with the state-action pair into the EQ.
- Evict an entry from the EQ and update the QVStore.
- Set the filled bit in the EQ entry.

**Legend**

- S1, S2, S3, S4: States
- A1, A2, A3: Actions
- Max: The action with the maximum Q-Value

**Note**

- Details on the process and equations are provided in the text alongside the diagram.
Architecting QVStore

\[ S = \{ PC+\text{Delta}, \text{Sequence of last-4 deltas} \} \]

Find the Action with max Q-Value

State Vector

Look up QVStore

Q-Value Store (QVStore)

Generate prefetch
Architecting QVStore

Fast prefetch prediction

Fast retrieval of Q-values from QVStore

Efficient storage organization of Q-values in QVStore

S = \{PC+Delta, Sequence of last-4 deltas\}

Q Value Store
Organization of QVStore

- A **monolithic** two-dimensional table?
  - Indexed by state and action values
- State-space increases **exponentially** with #bits

\[ S = \{ \text{PC+Delta, Sequence of last-4 deltas} \} \]

- \[ 32b + 7b + 4 \times 7b = 67 \text{ bits} \]
- \[ 2^{67} \text{ states} \]

- 127 actions

**SAFARI**

Design complexity

Access latency
Organization of QVStore

• We partition QVStore into $k$ vaults \([k = \text{number of features in state}]\)
  - Each vault corresponds to one feature and stores the Q-values of feature-action pairs

To retrieve $Q(S,A)$ for each action

• Query each vault in parallel with feature and action
• Retrieve feature-action Q-value from each vault
• Compute $\text{MAX}$ of all feature-action Q-values

$\text{MAX}$ ensures the $Q(S,A)$ is driven by the constituent feature that has highest $Q(\phi,A)$
Organization of QVStore

- We further partition each vault into multiple **planes**
  - Each plane stores a **partial** Q-value of a feature-action pair

To retrieve $Q(\phi,A)$ for each action

- Query each plane in **parallel** with hashed feature and action
- Retrieve partial feature-action Q-value from each plane
- Compute **SUM** of all partial feature-action Q-values
Organization of QVStore

• We further partition each vault into multiple **planes**
  - Each plane stores a **partial** Q-value of a feature-action pair

1. **Enables sharing** of partial Q-values between **similar** feature values, shortens prefetcher training time

2. **Reduces chances** of sharing partial Q-values across widely **different** feature values
More in the Paper

• **Pipelined search** operation for QVStore

• Reward assignment and **QVStore update**

• **Automatic design-space exploration**
  - Feature types
  - Action
  - Reward and Hyperparameter values
More in the Paper

- Pipelined search operation for QVStore

- Reward assignment and QVStore update

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

Talk Outline

- Key Shortcomings of Prior Prefetchers
- Formulating Prefetching as Reinforcement Learning
- Pythia: Overview
- Evaluation of Pythia and Key Results
- Conclusion
Simulation Methodology

- **Champsim** [3] trace-driven simulator

- **150** single-core memory-intensive workload traces
  - SPEC CPU2006 and CPU2017
  - PARSEC 2.1
  - Ligra
  - Cloudsuite

- Homogeneous and heterogeneous multi-core mixes

- **Five** state-of-the-art prefetchers
  - SPP [Kim+, MICRO’16]
  - Bingo [Bakhshalipour+, HPCA’19]
  - MLOP [Shakerinava+, 3rd Prefetching Championship, 2019]
  - SPP+DSPatch [Bera+, MICRO’19]
  - SPP+PPF [Bhatia+, ISCA’20]

Basic Pythia Configuration

• Derived from **automatic design-space exploration**

**State:** 2 features
- PC+Delta
- Sequence of last-4 deltas

**Actions:** 16 prefetch offsets
- Ranging between -6 to +32. Including 0.

**Rewards:**
- $R_{AT} = +20; R_{AL} = +12; R_{NP-H} = -2; R_{NP-L} = -4$
- $R_{IN-H} = -14; R_{IN-L} = -8; R_{CL} = -12$

SAFARI
Performance with Varying Core Count

Geomean speedup over no prefetching

Number of cores

- **Pythia**: 3.4% increase
- **SPP**: 7.7% increase
- **MLOP**
- **Bingo**

SAFARI
1. Pythia consistently provides the highest performance in all core configurations

2. Pythia’s gain increases with core count
Performance with Varying DRAM Bandwidth

The graph shows the geometric speedup of performance over no prefetching against DRAM MTPS (in log scale). The lines represent different benchmarks and processors: Pythia, Bingo, MLOP, SPP, Baseline, ~Intel Xeon 6258R, ~AMD EPYC Rome 7702P, and ~AMD Threadripper 3990x.

Key points:
- The green line with purple shading for SPP shows a 17% speedup at 200 MTPS.
- The red line with yellow shading for Pythia shows a 3% speedup at 12800 MTPS.
- The blue line with green shading for ~AMD Threadripper 3990x shows a significant increase in speedup as DRAM MTPS increases.

The x-axis represents DRAM MTPS (in log scale), while the y-axis represents the geometric speedup over no prefetching.
Pythia outperforms prior best prefetchers for a wide range of DRAM bandwidth configurations.
Performance Improvement via Customization

- **Reward value customization**
- **Strict Pythia configuration**
  - Increasing the rewards for no prefetching
  - Decreasing the rewards for inaccurate prefetching

- Strict Pythia is more conservative in generating prefetch requests than the basic Pythia
- Evaluate on all Ligra graph processing workloads
Performance Improvement via Customization

IPC normalized to no prefetching

- PageRank: 3.1%
- PageRankDelta: 2.8%
- CC: 3.4%
- BFS: 7.8%
- BC: 5.2%
- GEOMEAN: 2%

SAFARI
Performance Improvement via Customization

Pythia can extract even higher performance via customization **without changing hardware**
Pythia’s Overhead

• **25.5 KB** of total metadata storage *per core*
  - Only simple tables

• We also model functionally-accurate Pythia with full complexity in Chisel [4] HDL

- 1.03% area overhead
- 0.4% power overhead
- Satisfies prediction latency

of a desktop-class 4-core Skylake processor (Xeon D2132IT, 60W)

More in the Paper

• Performance comparison with **unseen traces**
  - Pythia provides **equally high** performance benefits

• Comparison against **multi-level prefetchers**
  - Pythia **outperforms** prior best multi-level prefetchers

• Understanding Pythia’s learning with **a case study**
  - We reason towards **the correctness** of Pythia’s decision

• **Performance sensitivity** towards different features and hyperparameter values

• Detailed single-core and four-core performance
More in the Paper

- Performance comparison with unseen traces
  - Pythia provides equally high performance benefits

- Comparison against multi-level prefetchers
  - Pythia outperforms prior best multi-level prefetchers

- Understanding Pythia's learning with a case study
  - We reason towards the correctness of Pythia's decision

- Performance sensitivity towards different features and hyperparameter values

- Detailed single-core and four-core performance

---

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

Pythia is Open Source

https://github.com/CMU-SAFARI/Pythia

- MICRO’21 artifact evaluated
- Champsim source code + Chisel modeling code
- All traces used for evaluation
Executive Summary

- **Background**: Prefetchers predict addresses of future memory requests by associating memory access patterns with program context (called **feature**).

- **Problem**: Three key shortcomings of prior prefetchers:
  - Predict mainly using a **single program feature**
  - Lack **inherent system awareness** (e.g., memory bandwidth usage)
  - Lack **in-silicon customizability**

- **Goal**: Design a prefetching framework that:
  - Learns from **multiple features** and **inherent system-level feedback**
  - Can be **customized in silicon** to use different features and/or prefetching objectives

- **Contribution**: Pythia, which formulates prefetching as reinforcement learning problem
  - Takes **adaptive** prefetch decisions using multiple features and system-level feedback
  - Can be **customized in silicon** for target workloads via simple configuration registers
  - Proposes a **realistic and practical** implementation of RL algorithm in hardware

- **Key Results**:
  - Evaluated using a wide range of workloads from SPEC CPU, PARSEC, Ligra, Cloudsuite
  - Outperforms best prefetcher (in 1-core config.) by **3.4%, 7.7% and 17%** in 1/4/bw-constrained cores
  - Up to **7.8% more performance** over basic Pythia across Ligra workloads via simple customization

**SAFARI**

[https://github.com/CMU-SAFARI/Pythia](https://github.com/CMU-SAFARI/Pythia)
Pythia
A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, Onur Mutlu

https://github.com/CMU-SAFARI/Pythia
BACKUP
Reward Assignment to EQ Entry

- **Every** action gets inserted into EQ
- Reward is assigned to each EQ entry **before or during** the eviction

- **During EQ insertion**: for actions
  - Not to prefetch
  - Out-of-page prefetch
Reward Assignment to EQ Entry

- **Every** action gets inserted into EQ
- Reward is assigned to each EQ entry **before or during** the eviction

- **During EQ insertion**: for actions
  - Not to prefetch
  - Out-of-page prefetch

- **During EQ residency**: 
  - In case address of a demand matches with address in EQ *(signifies accurate prefetch)*
Reward Assignment to EQ Entry

- **Every** action gets inserted into EQ
- Reward is assigned to each EQ entry **before or during** the eviction

- **During EQ insertion:** for actions
  - Not to prefetch
  - Out-of-page prefetch

- **During EQ residency:**
  - In case address of a demand matches with address in EQ
    (*signifies accurate prefetch*)

- **During EQ eviction:**
  - In case no reward is assigned till eviction
    (*signifies inaccurate prefetch*)
Performance S-curve: Single-core

- Speedup over no prefetching
- Workload number
- Workloads: SPP, Bingo, MLOP, Pythia

- Benchmark examples:
  - SPP: 623.xalancbmk_s-592B
  - Bingo: 603.bwaves_s-2931B
  - MLOP: 462.libquantum
Performance S-curve: Four-core

Speedup over no prefetching

Workload number

- SPP
- Bingo
- MLOP
- Pythia

- 429.mcf-184B
- 462.libquantum-1343B
- 437.leslie3d-271B
- raytrace-23.75B
- Mix-59
- Mix-240
- pagerank