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

SAFARI Research Group
safari.ethz.ch

1 Mainly use **one** program context info. for prediction

2 Lack **inherent system awareness**

3 Lack **in-silicon customizability**

Why do prefetchers not perform well?

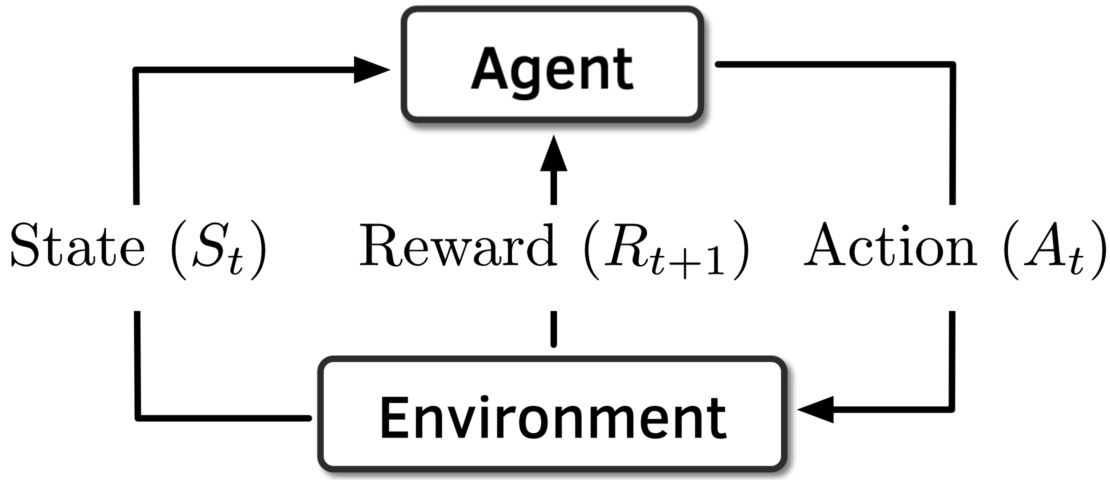
Pythia

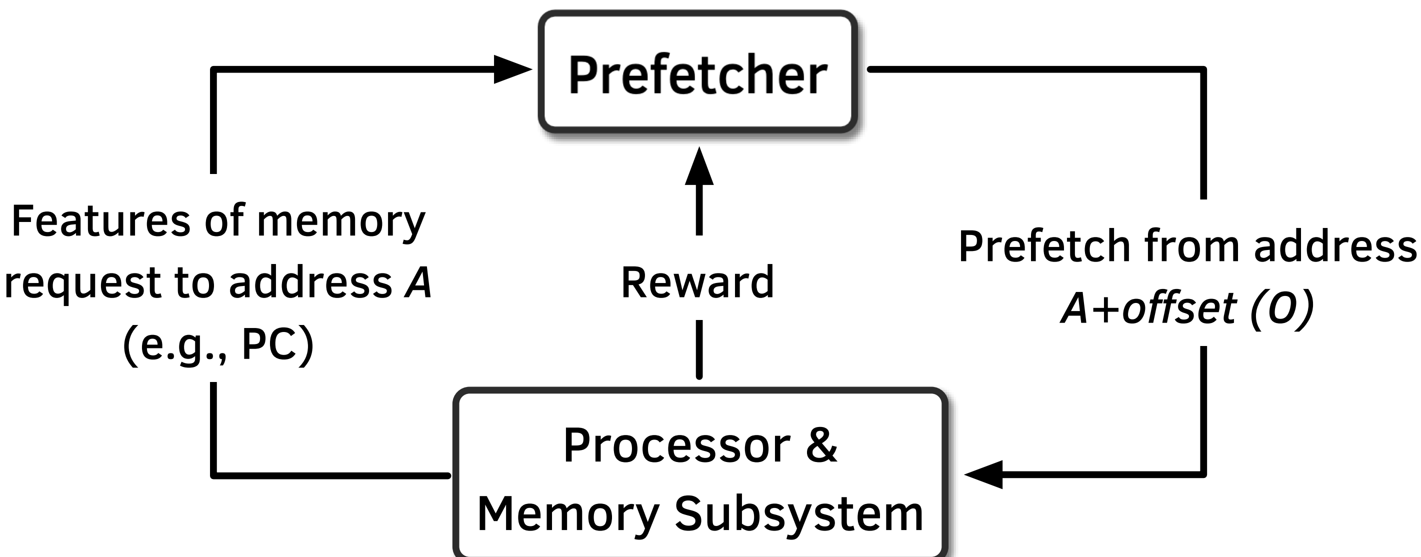
Autonomously learns to prefetch using **multiple program context information** and **system-level feedback**

Can be **customized in silicon** to change program context information or prefetching objective on the fly

Brief Overview of Pythia

Pythia formulates prefetching as a **reinforcement learning** problem

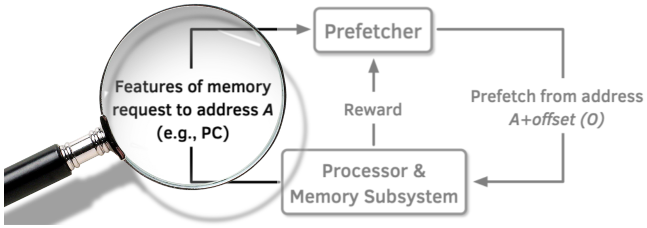




What is State?

- ***k*-dimensional** vector of features

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

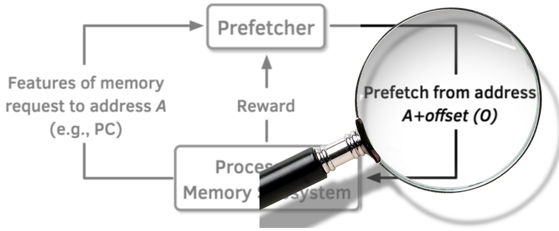


- Feature = control-flow + data-flow

- **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 Action?

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



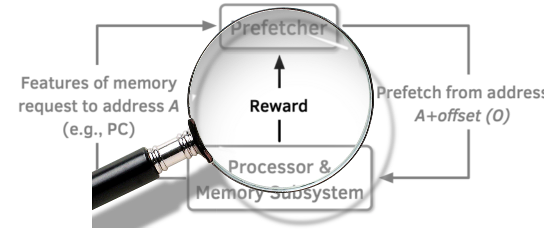
- **Action-space**: 127 actions in the range [-63, +63]
  - For a machine with 4KB page and 64B cacheline

- 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?

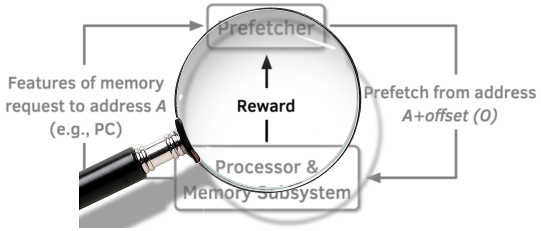


- Defines the **objective** of Pythia

- 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

Simulation Methodology

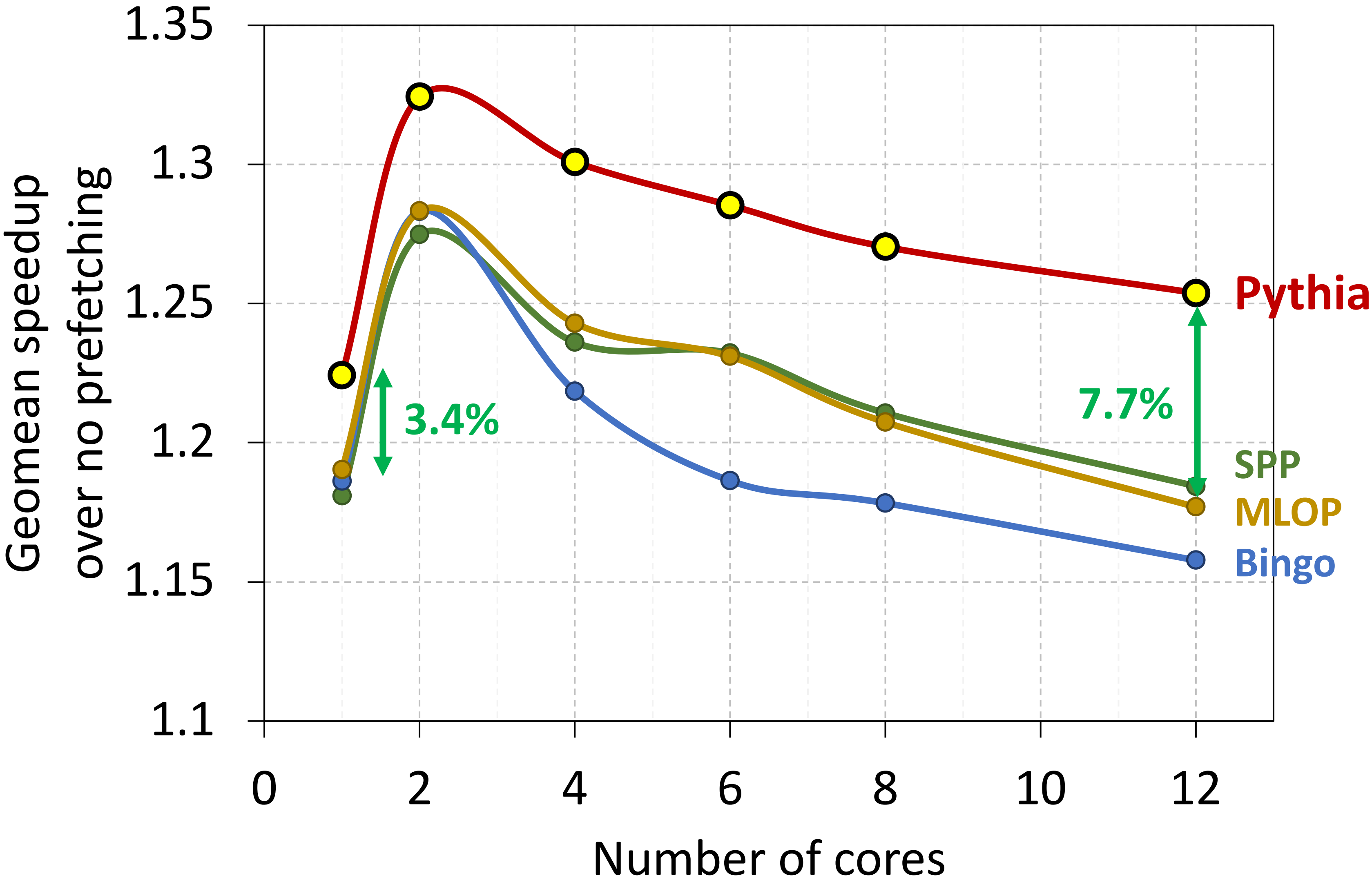
- **Champsim** [3] trace-driven simulator

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

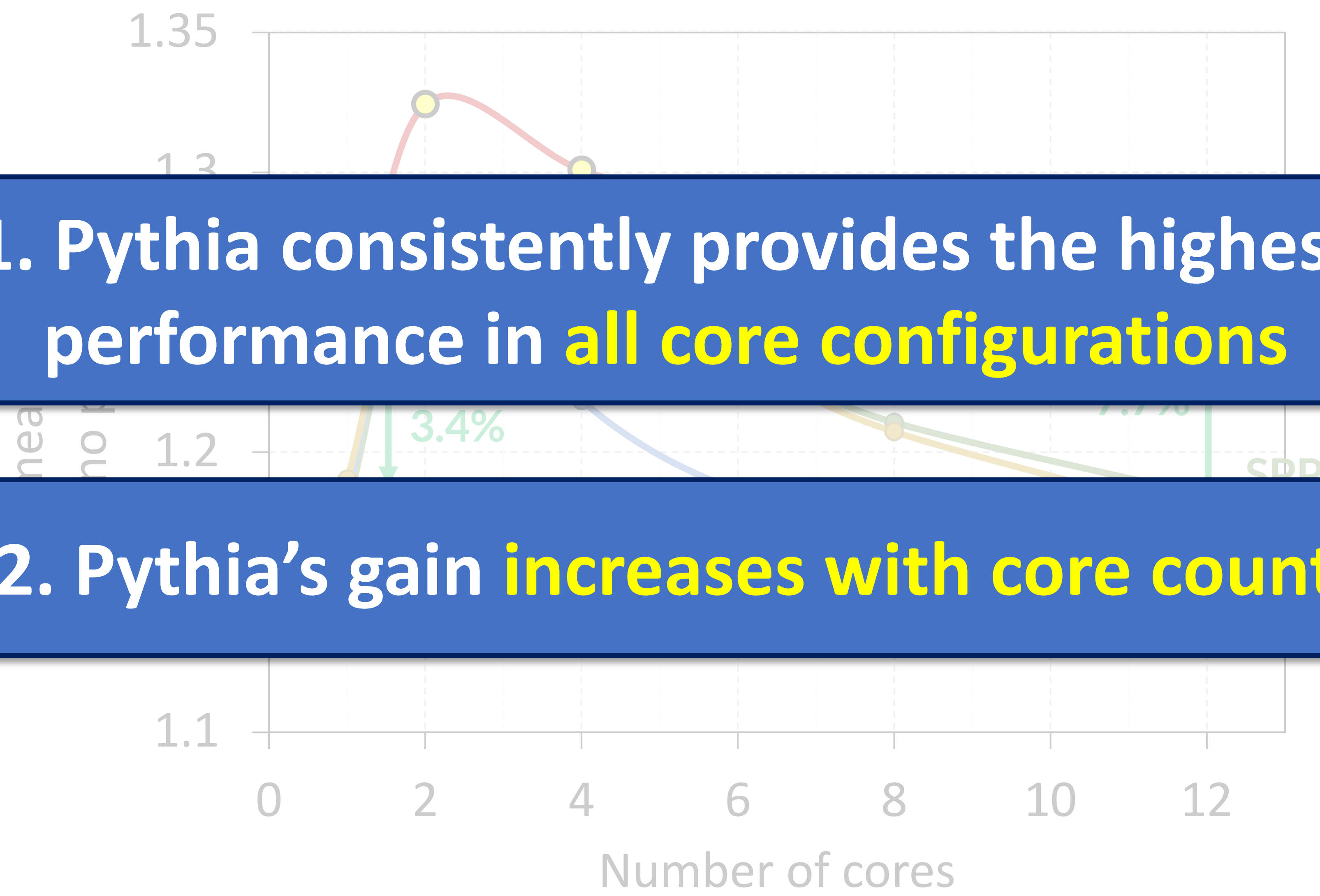
- **Five** state-of-the-art prefetchers
  - SPP [Kim+, MICRO'16]
  - Bingo [Bakhshalipour+, HPCA'19]
  - MLOP [Shakerinava+, 3rd Prefetching Championship, 2019]
  - SPP+DSPatch [Bera+, MICRO'19]
  - SPP+PPF [Bhatia+, ISCA'20]

[3] https://github.com/ChampSim/ChampSim

Performance with Varying Core Count



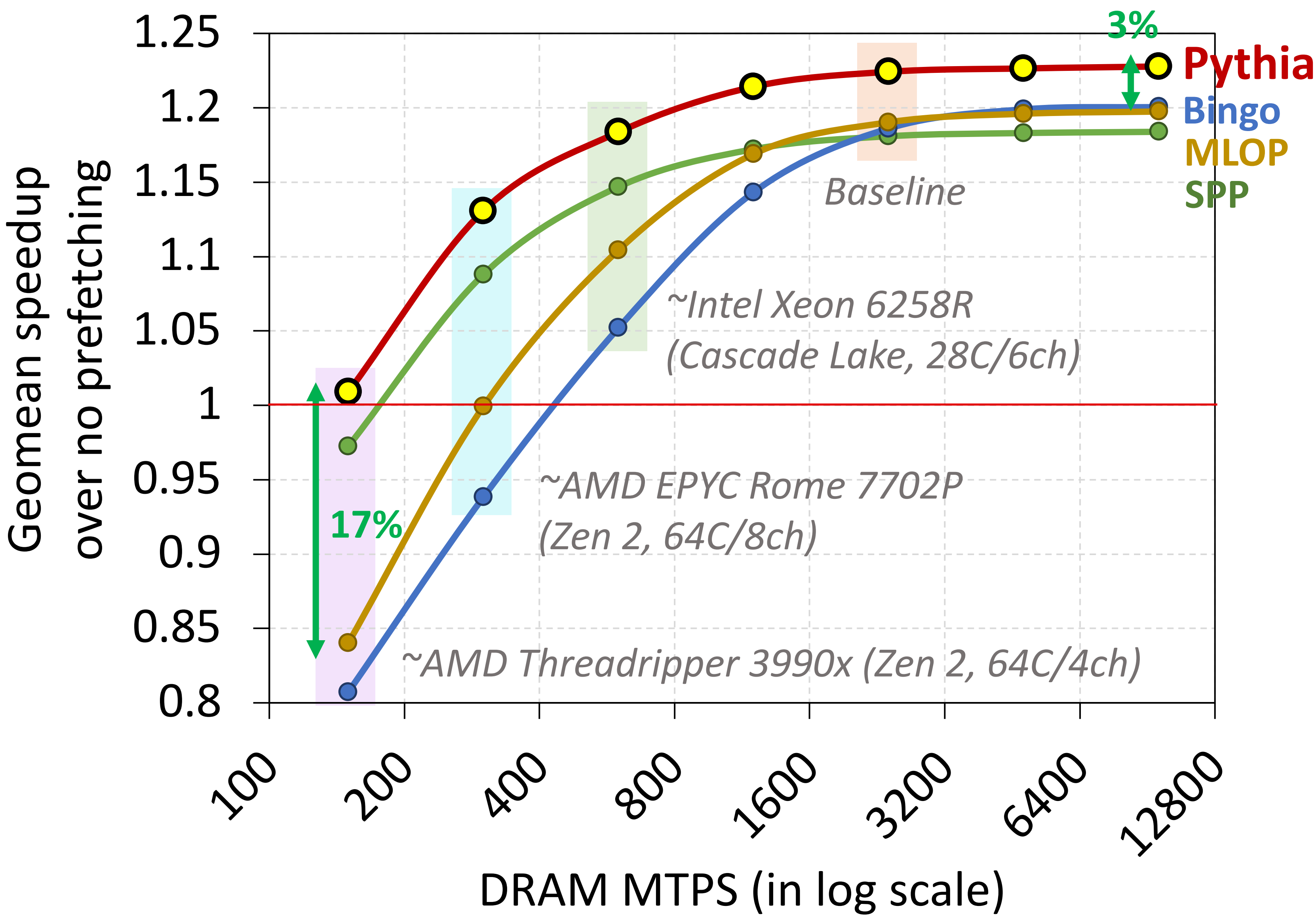
Performance with Varying Core Count



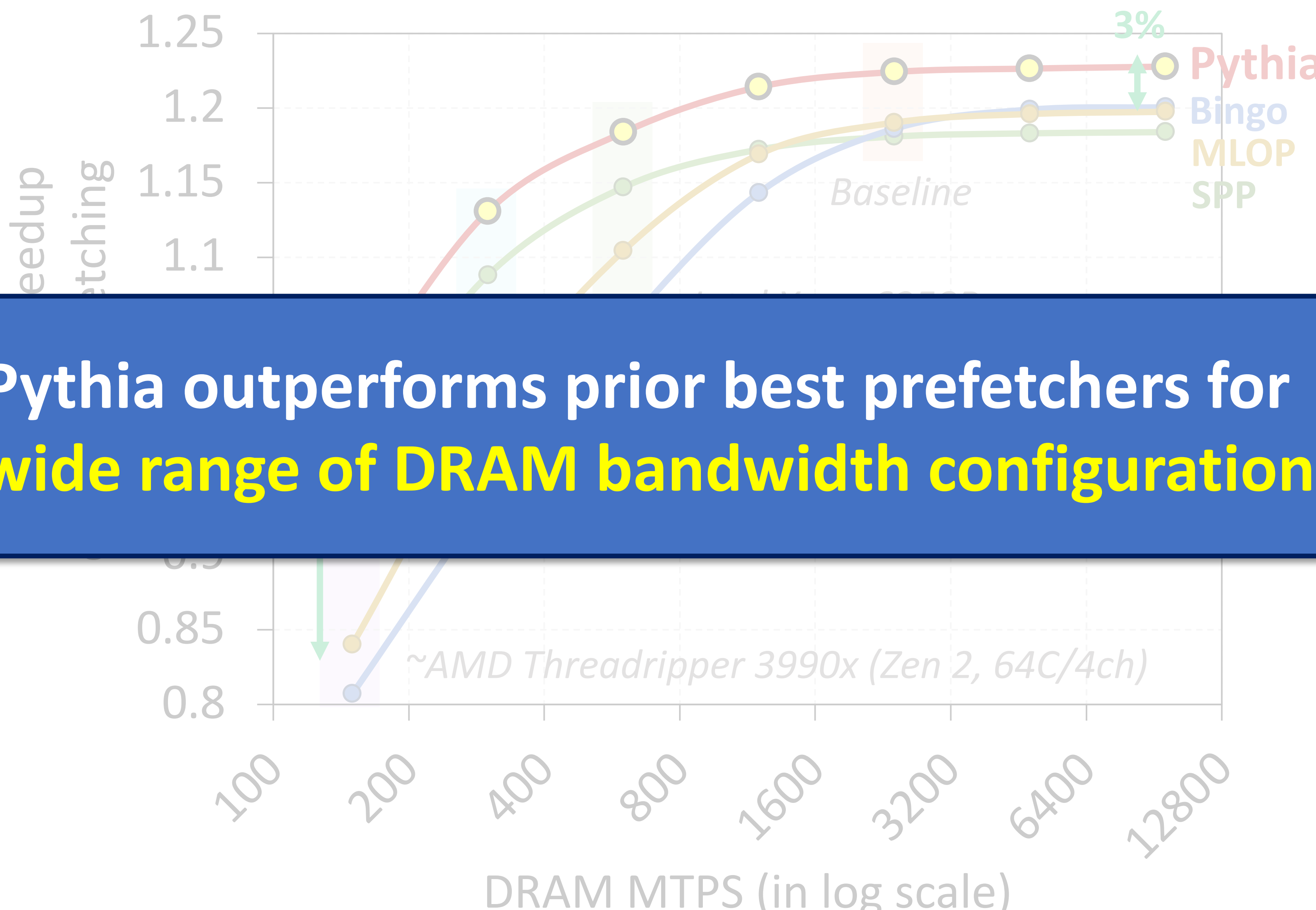
1. Pythia consistently provides the highest performance in **all core configurations**

2. Pythia's gain **increases with core count**

Performance with Varying DRAM Bandwidth



Performance with Varying DRAM Bandwidth



Pythia outperforms prior best prefetchers for a wide range of DRAM bandwidth configurations

Performance Improvement via Customization

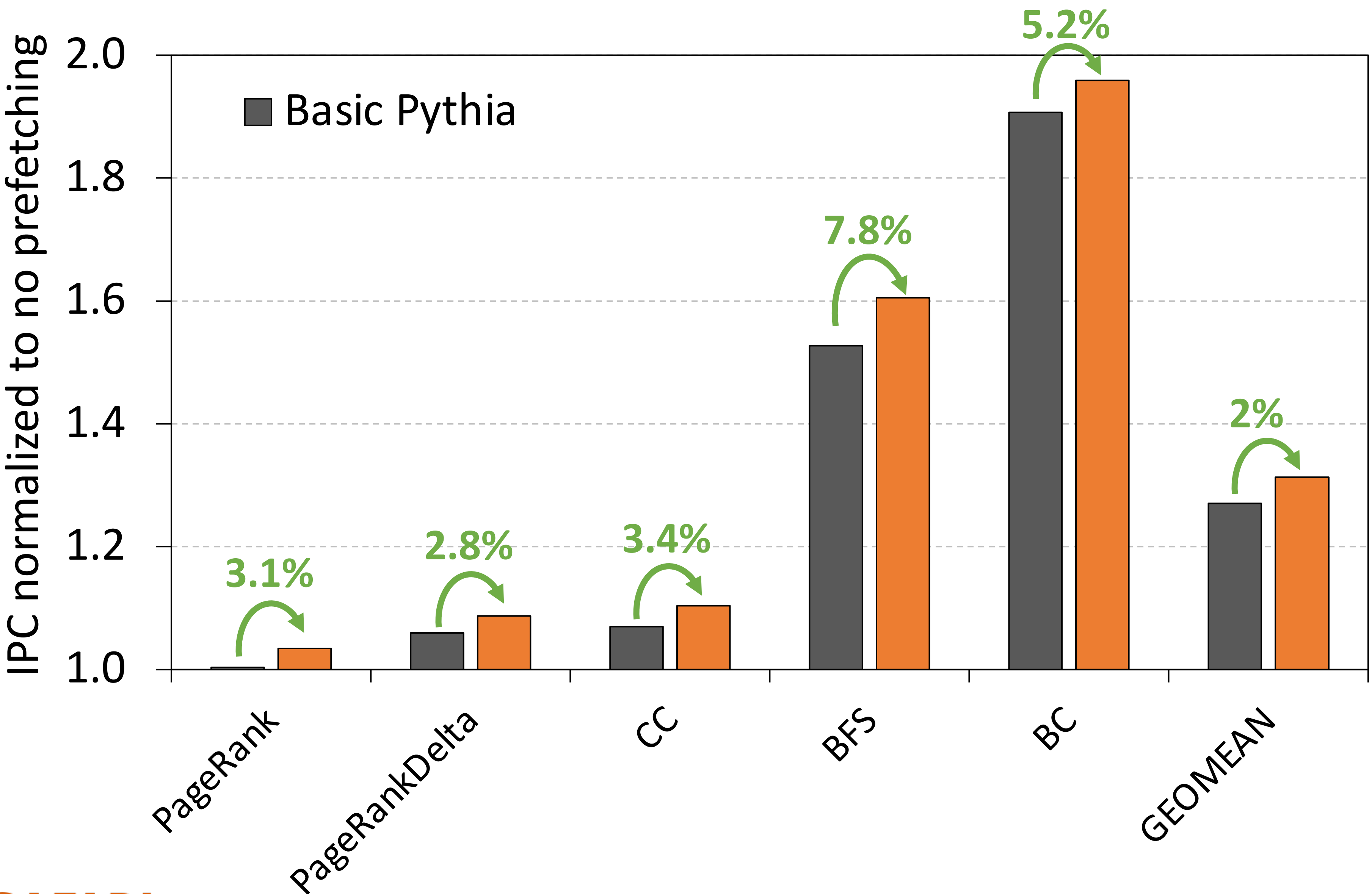
- Reward value customization

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

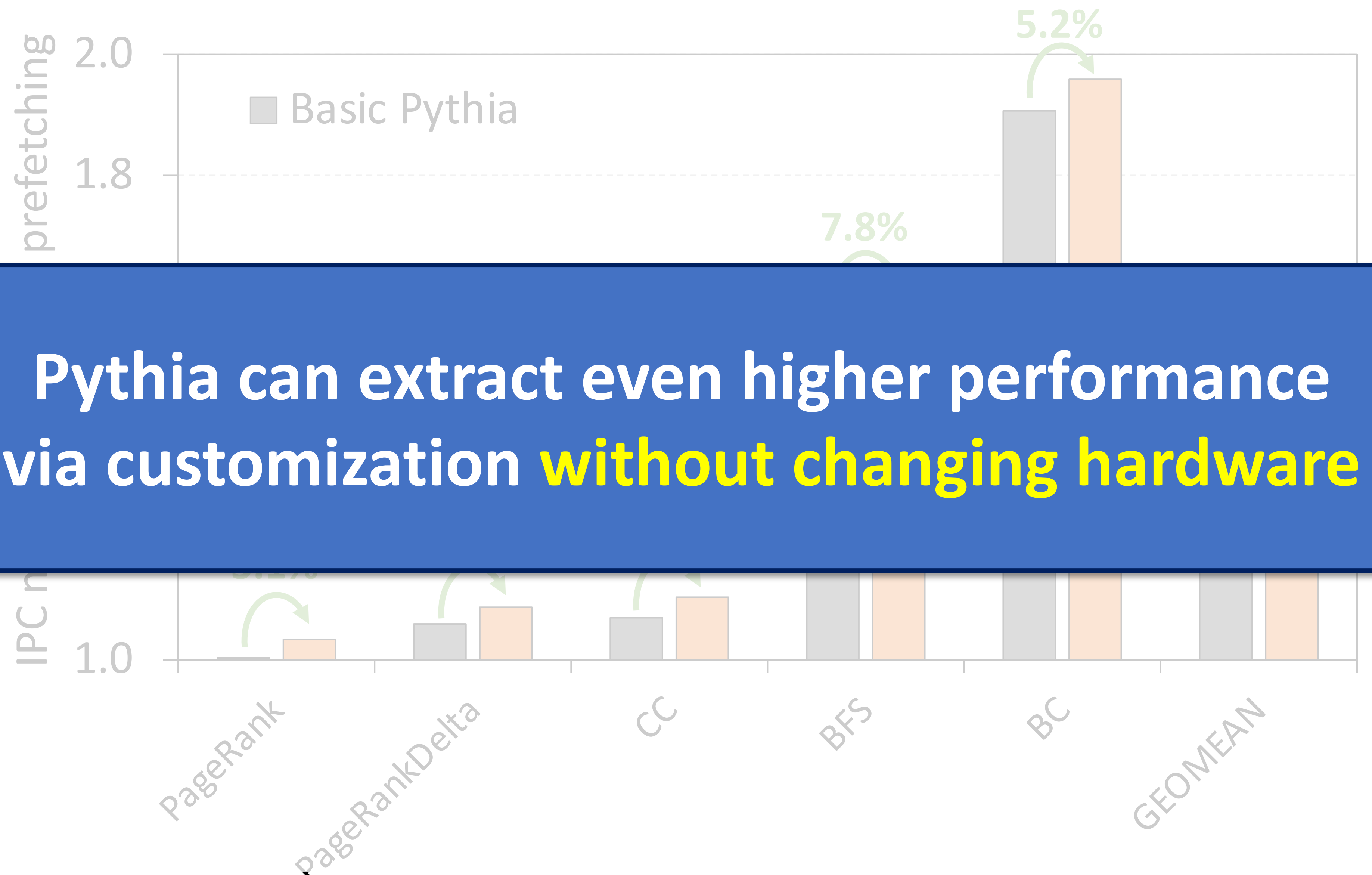
- Strict Pythia is **more conservative** in generating prefetch requests than the basic Pythia

- Evaluate on all **Ligra graph processing workloads**

Performance Improvement via Customization



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)

[4] https://www.chisel-lang.org

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: 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

https://arxiv.org/pdf/2109.12021.pdf

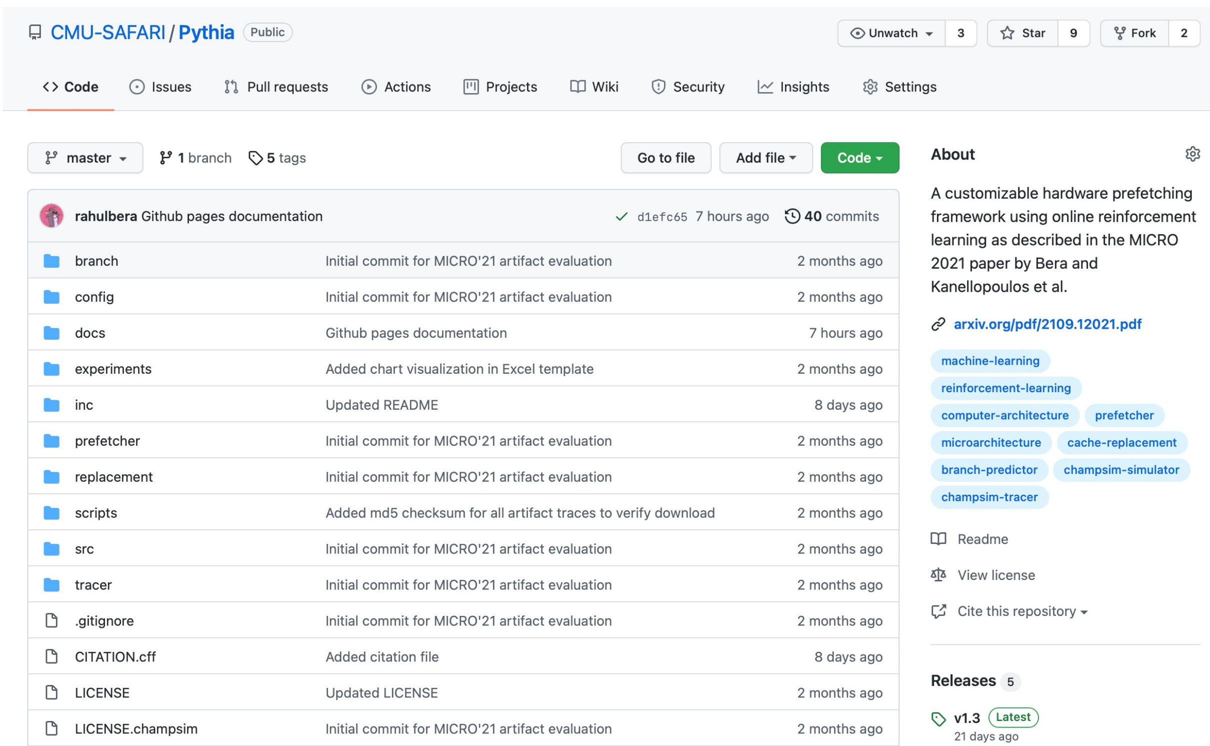
- Performance sensitivity towards different features and hyperparameter values

- Detailed single-core and four-core performance

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



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

SAFARI
SAFARI Research Group
safari.ethz.ch
ETH zürich
intel
TU Delft

Discussion

- **FAQs**
  - Why RL?
  - What about large page?
  - What's the prefetch degree?
  - Can customization happen during workload execution?
  - Can runtime mixing create problem?

- **Simulation and Methodology**
  - Basic Pythia configuration
  - System parameters
  - Configuration of prefetchers
  - Evaluated workloads
  - Feature selection

- **Detailed Design**
  - Reward structure
  - Design overview
  - QVStore Organization

- **More Results**
  - Comparison against other adaptive prefetchers
  - Comparison against Context prefetcher
  - Feature combination sensitivity
  - Hyperparameter sensitivity
  - Comparison with multi-level prefetchers
  - Performance in unseen workloads
  - Single-core s-curve
  - Four-core s-curve
  - Detailed performance analysis
  - Benefit of bandwidth awareness
  - Case study
  - Customizing rewards
  - Customizing features

FAQs

Why RL? Why Not Supervised Learning?

- Determining the **benefits of prefetching** (i.e., whether a decision was good for performance or not) is **not easy**
  - **Depends on a complex set of metrics**
    - Coverage, accuracy, timeliness
    - Effects on system: b/w usage, pollution, cross-application interference, …
  - **Dynamically-changing environmental conditions** change the benefit
  - **Delayed feedback due to long latency** (might not receive feedback at all for inaccurate prefetches!)

- Differs from classification tasks (e.g., branch prediction)
  - Performance strongly correlates mainly to accuracy
  - Does not depend on environment
  - Bounded feedback delay

What About Large Pages?

- Pythia's framework can be **easily extended** to incorporate additional prefetch actions (i.e., possible prefetch offsets for the page size)

- To decrease the storage overhead
  - **Prune action space** via automatic design-space exploration
  - **Hash action values** to retrieve Q-values

What is the Prefetch Degree? Is It Managed by the RL Agent?

- Pythia employs **a simple degree selector**, separate from the RL agent
  - If the agent has selected the same prefetch action (O) multiple times in a row, Pythia increases the degree (A+2O, A+3O, …)
  - At most degree 4

- Future works on managing degree by the RL agent

Can the Customization Be Done While the Workload is Running?

- Certainly.
- Pythia, being an **online learning** technique, will autonomously adapt (and optimize) its policy to use the new program features or the modified reward values

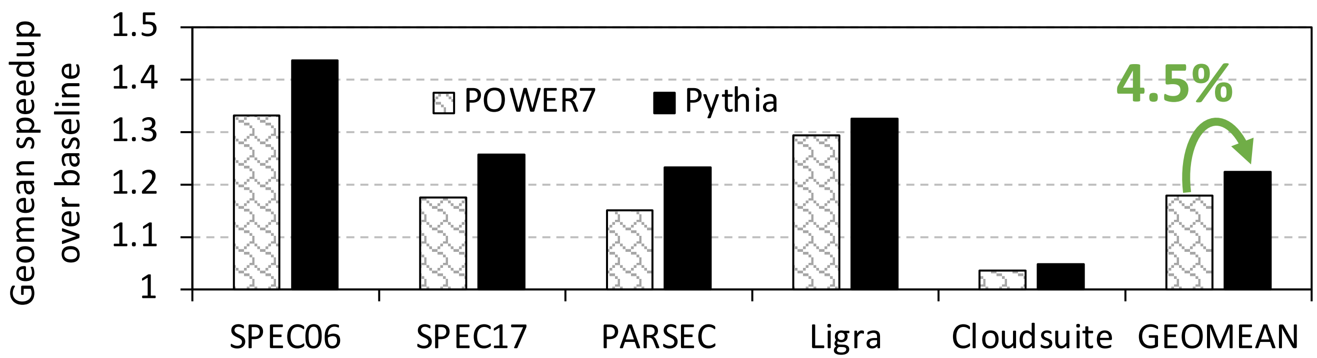
Can Runtime Workload Mix Create an Issue?

- We implement the bandwidth usage feedback using a counter in the memory controller. Thus Pythia already has a **global view of the memory bandwidth usage** that incorporates all workloads running on a multi-core system

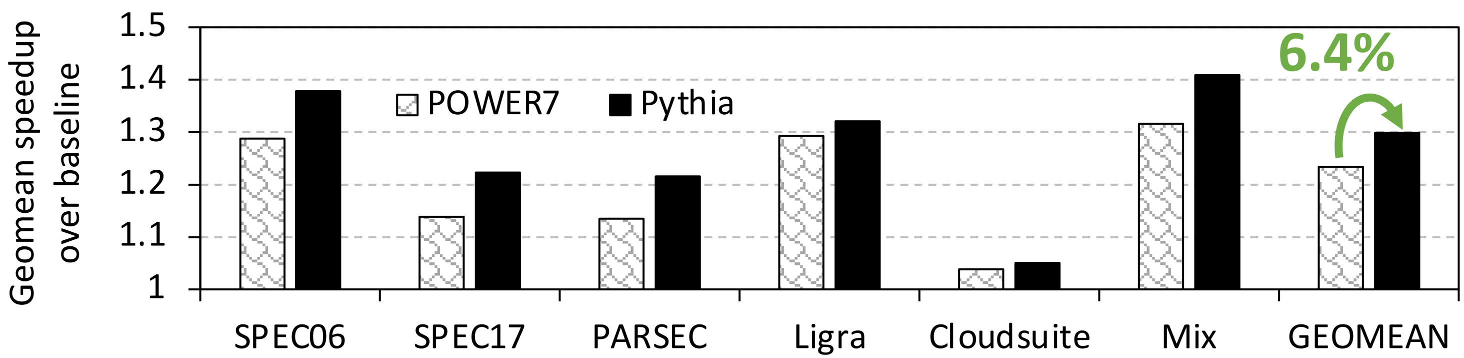
- We evaluate a diverse set (300 of each category) of four-core, eight-core, twelve-core random workload mixes
- Based on our evaluation, we observe that **Pythia dynamically adapts** itself to varying workload demands

How does Pythia Compare Against Other Adaptive Prefetching Solutions?

- We compare Pythia against **IBM POWER7**[5] prefetcher
  - Adaptively selects prefetcher degree/configuration by monitoring program IPC



(a) single-core

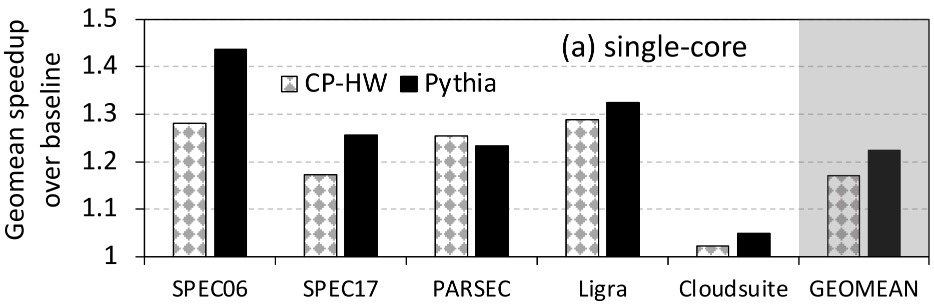


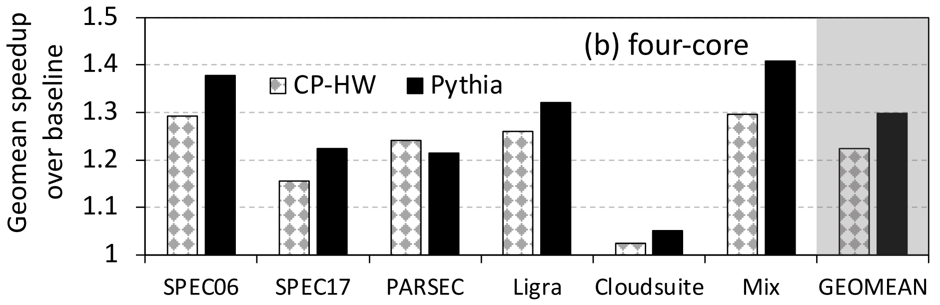
(b) four-core

[5] Jimenez et al., TOPC'14

How Does Pythia Compare Against the Context Prefetcher?

- Pythia widely differs from the Context Prefetcher (CP)[6] in all three aspects: state, action, and reward. The key differences are:
  - CP **does not consider system-level feedback**
  - CP models the agent as a contextual bandit which **takes myopic prefetch decisions** as compared to Pythia
  - CP **requires compiler support** to extract software-level features



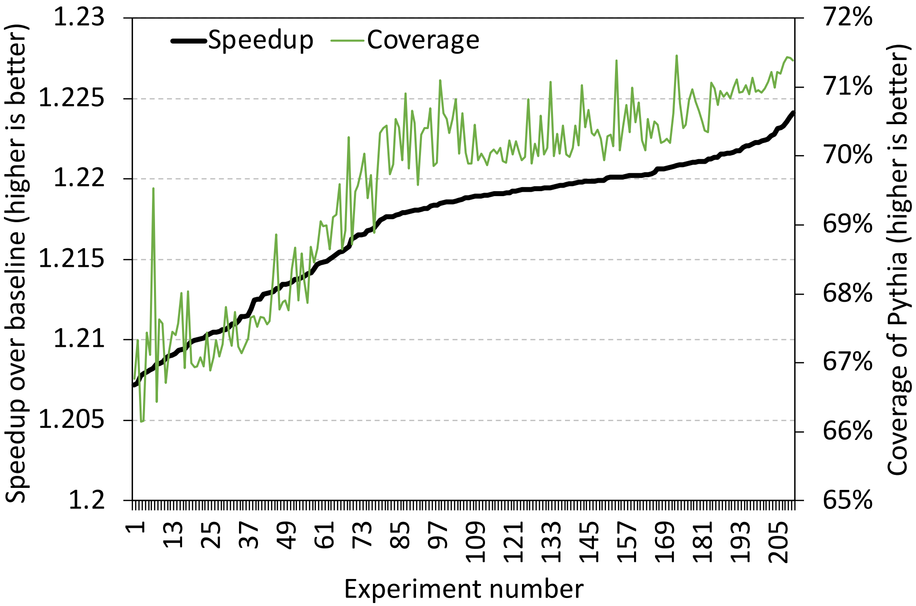


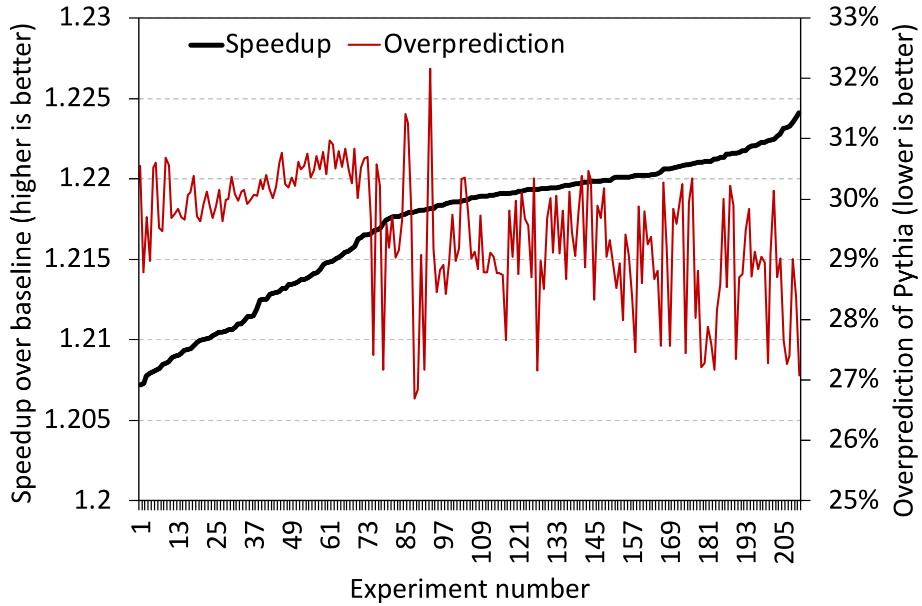
Pythia outperforms CP-HW by 5.3% in single-core and 7.6% in four-core system

[6] Leeor et al., ISCA'15

How Pythia's Performance Changes With Various State Definitions You Have Swept?

- In total we evaluate state defined as any-one, any-two, and any-three combinations of 32 features





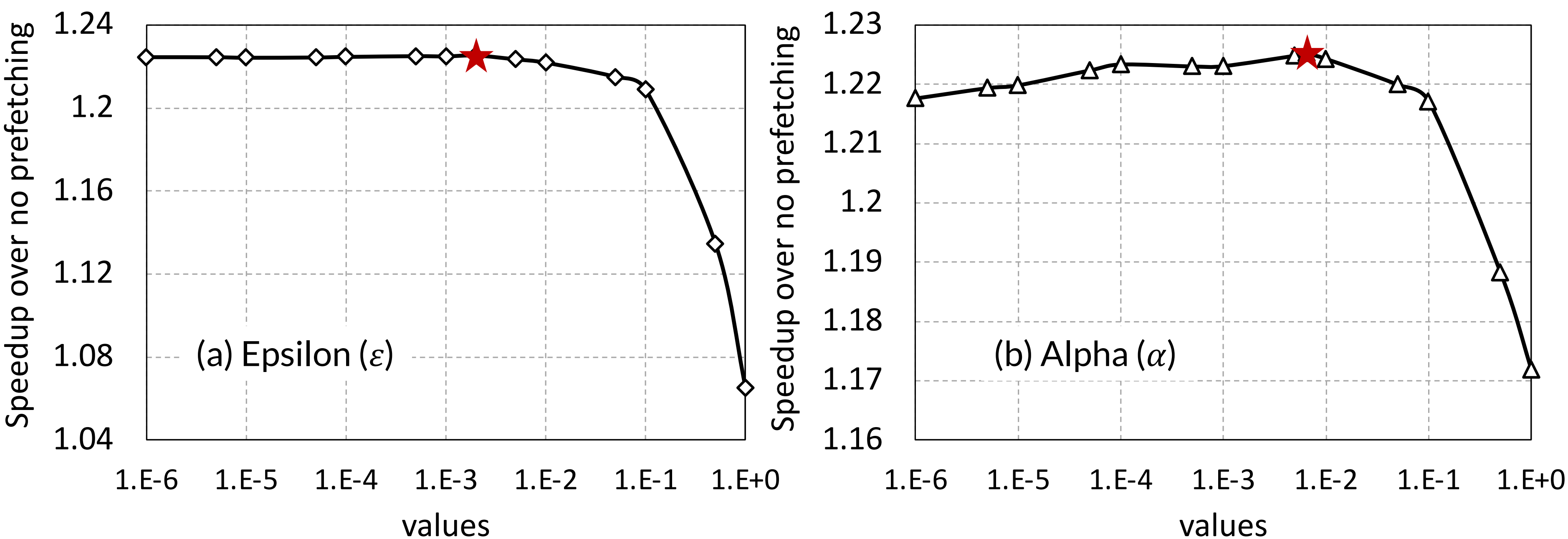
Performance gain ranges from 20.7% to 22.4%

Coverage ranges from 66.2% to 71.5%

Overprediction ranges from 26.7% to 32.2%

Is Pythia Sensitive to Hyperparameter?

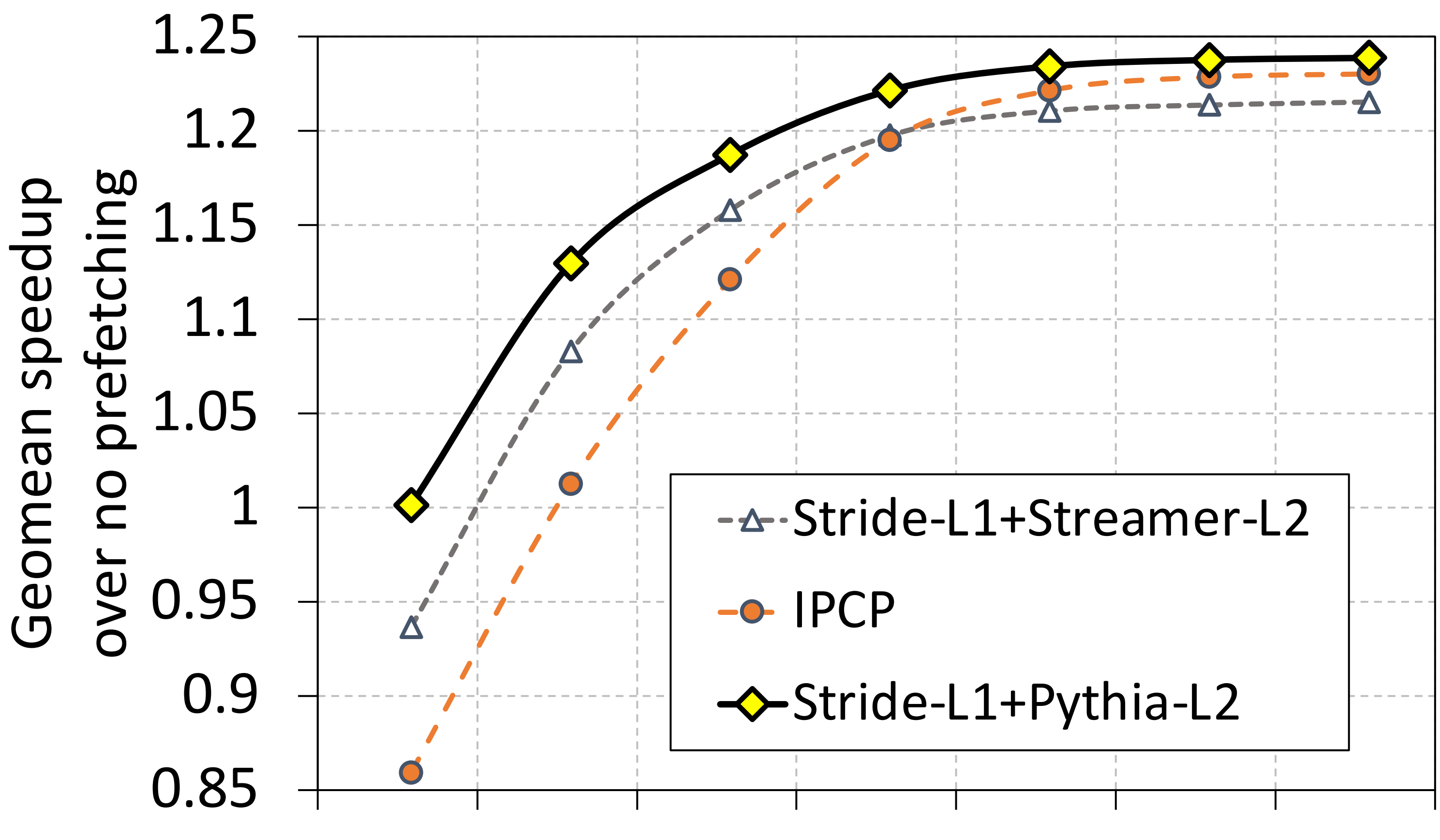
- Not setting hyperparameters can significantly impact the overall performance improvement



Changing ε from 0.002 to 1.0 drops perf. by 16%

Changing α from 0.0065 to 1.0 drops perf. by 5.4%

How Does Pythia Compare Against Commercial Multi-level Prefetchers?

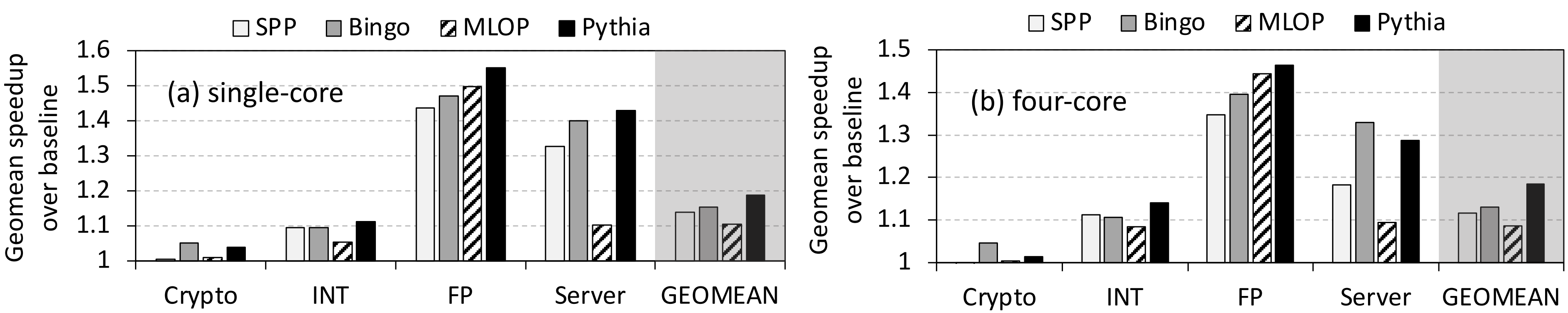


Pythia outperforms IPCP [7] by 14.2% on average in 150-MTPS

DRAM MTPS (in log scale)

Does Pythia Perform Equally Well for Unseen Workloads also?

- Evaluated with 500 traces from value prediction championship
  - **No prefetcher has been trained on these traces**



Pythia outperforms MLOP and Bingo by 8.3% and 3.5% in single-core

And 9.7% and 5.4% in four-core

Basic Pythia Configuration

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

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

System Parameters

Table 5: Simulated system parameters

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

Configuration of Prefetchers

Table 7: Configuration of evaluated prefetchers

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

Evaluated Workloads

Table 6: Workloads used for evaluation

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

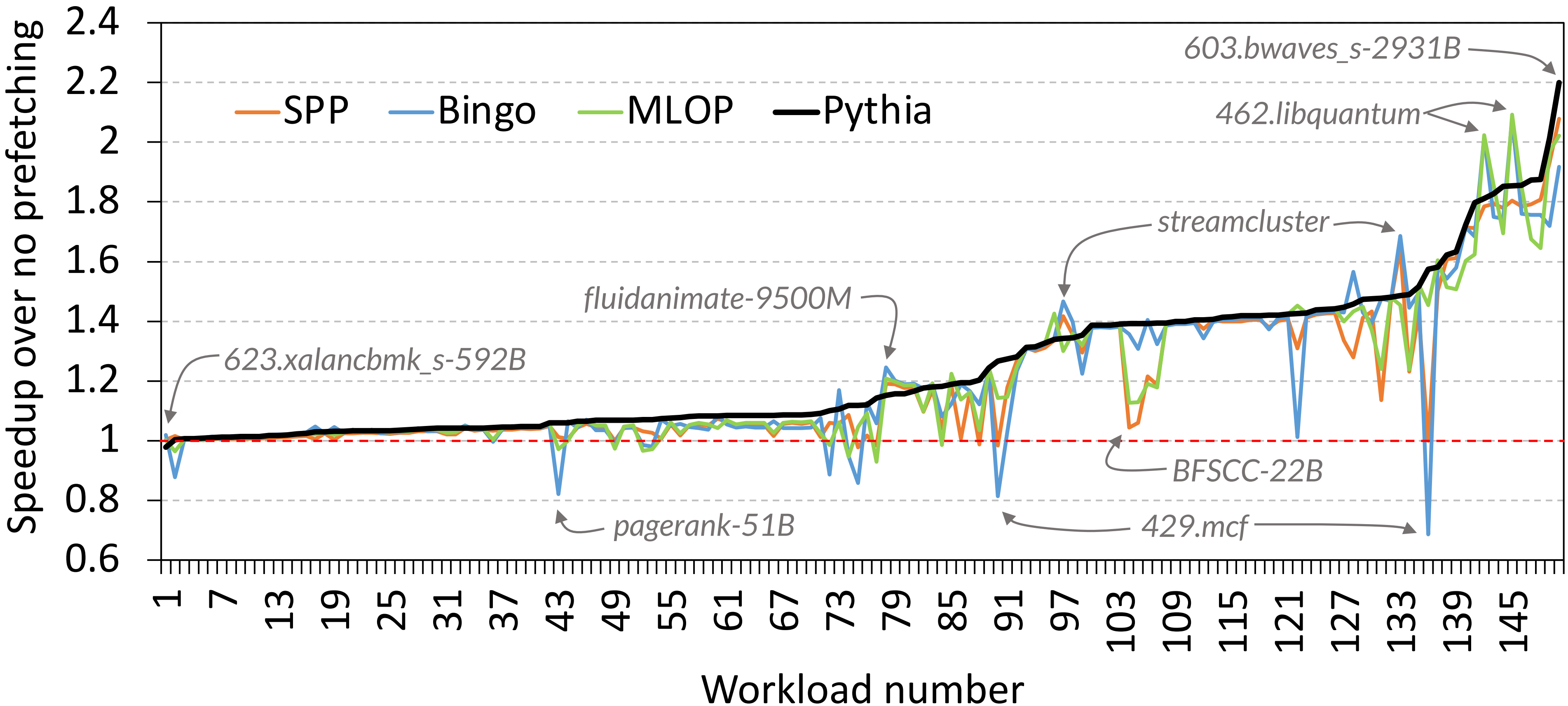
List of Evaluated Features

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

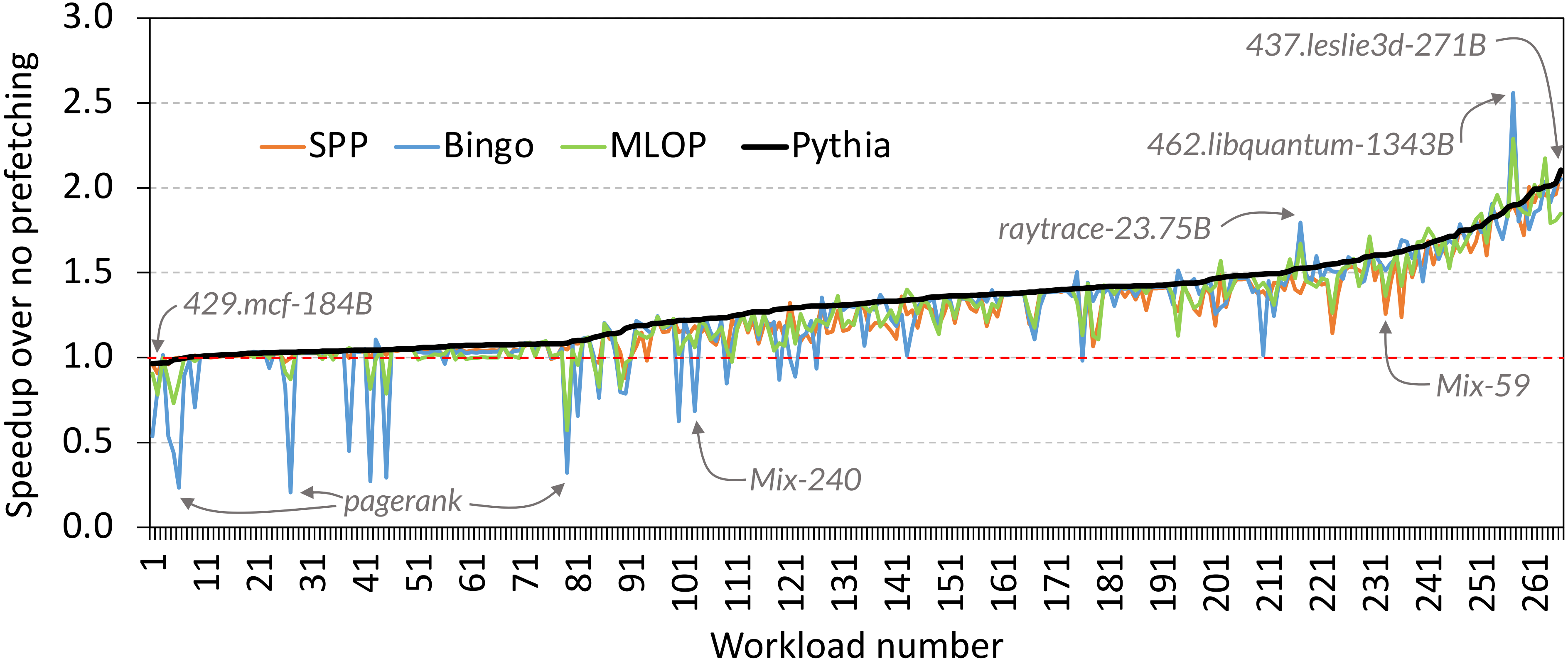
Control-flow Component	Data-flow Component
(1) PC of load request (2) PC-path (XOR-ed last-3 PCs) (3) PC XOR-ed branch-PC (4) None	(1) Load cacheline address (2) Page number (3) Page offset (4) Load address delta (5) Sequence of last-4 offsets (6) Sequence of last-4 deltas (7) Offset XOR-ed with delta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

MORE RESULTS

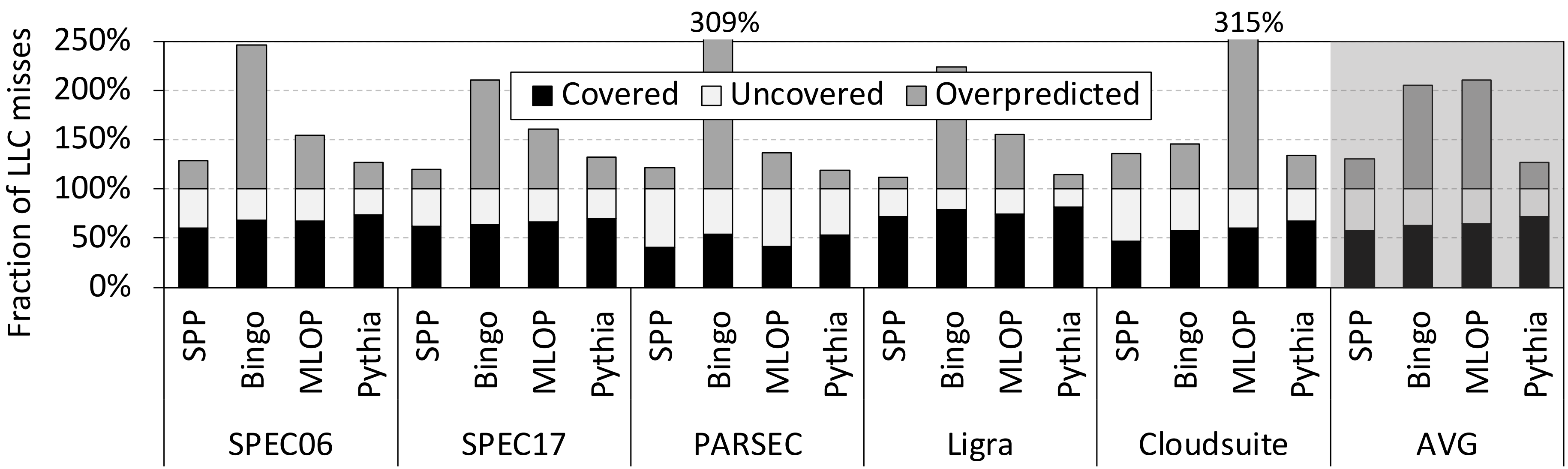
Performance S-curve: Single-core



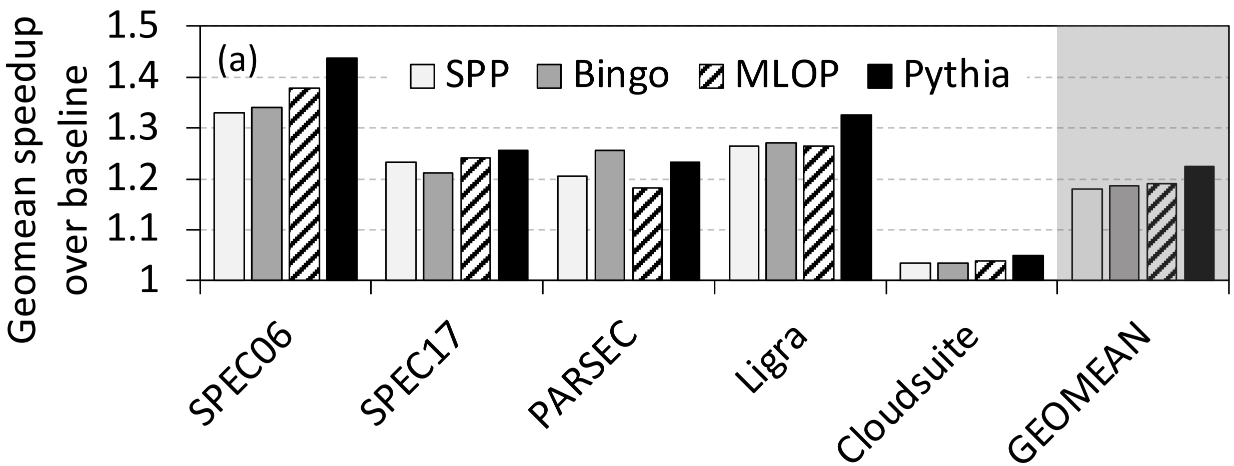
Performance S-curve: Four-core

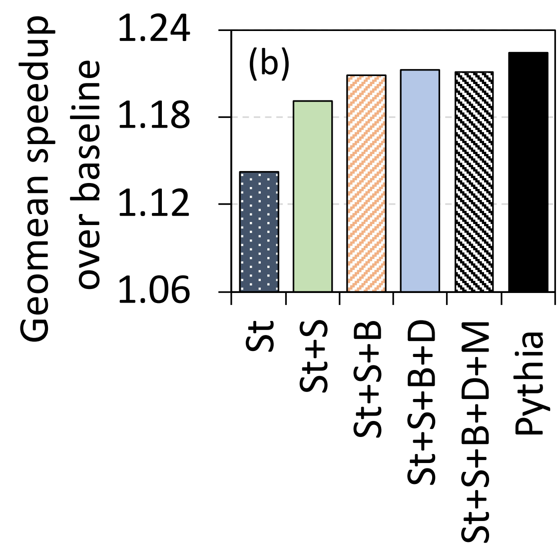


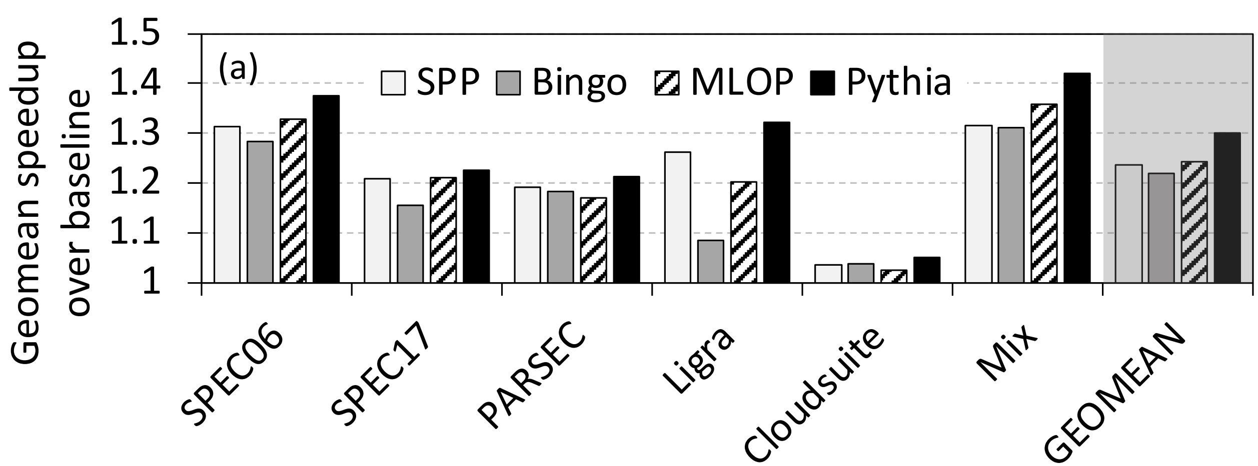
Single-core Coverage & Overprediction

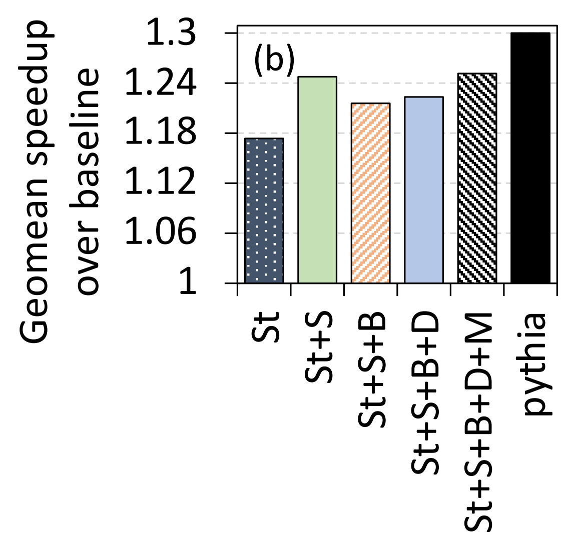


Detailed Performance

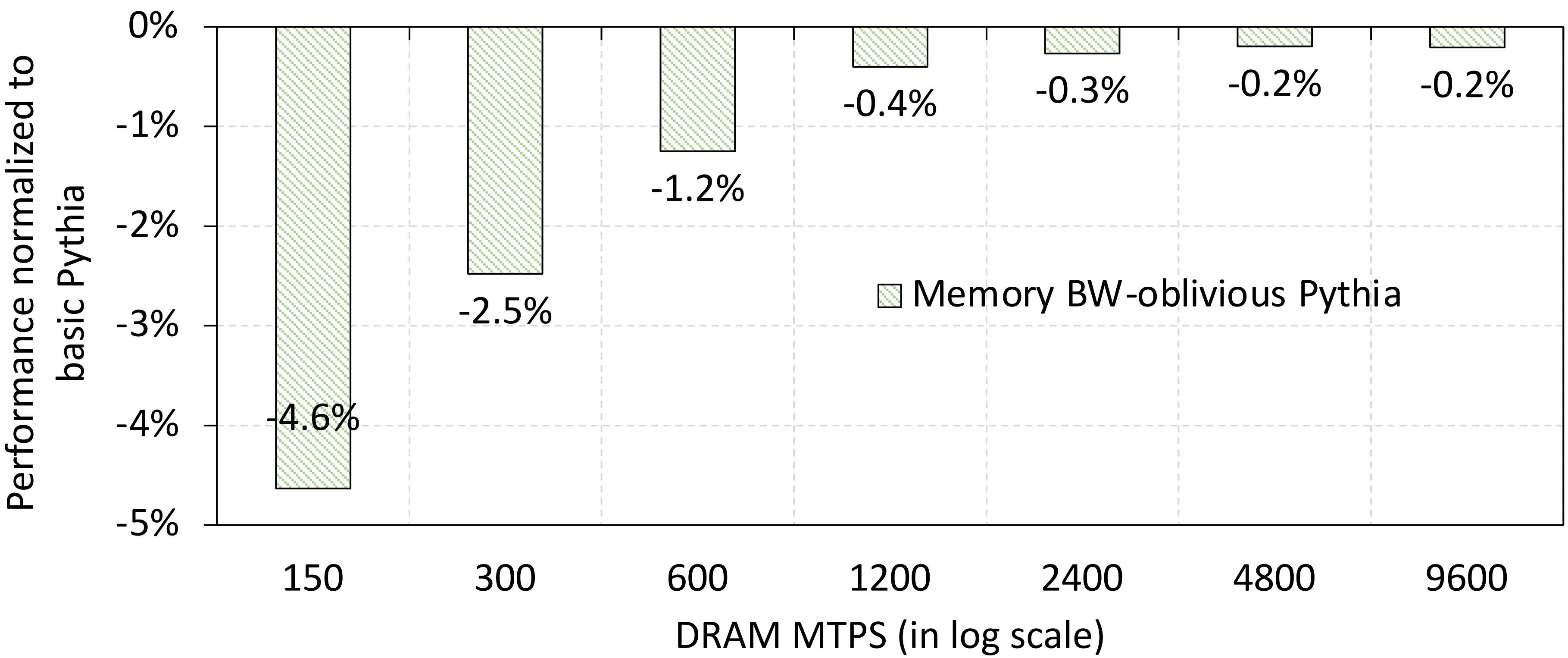








Benefit of Bandwidth Awareness



Case Study

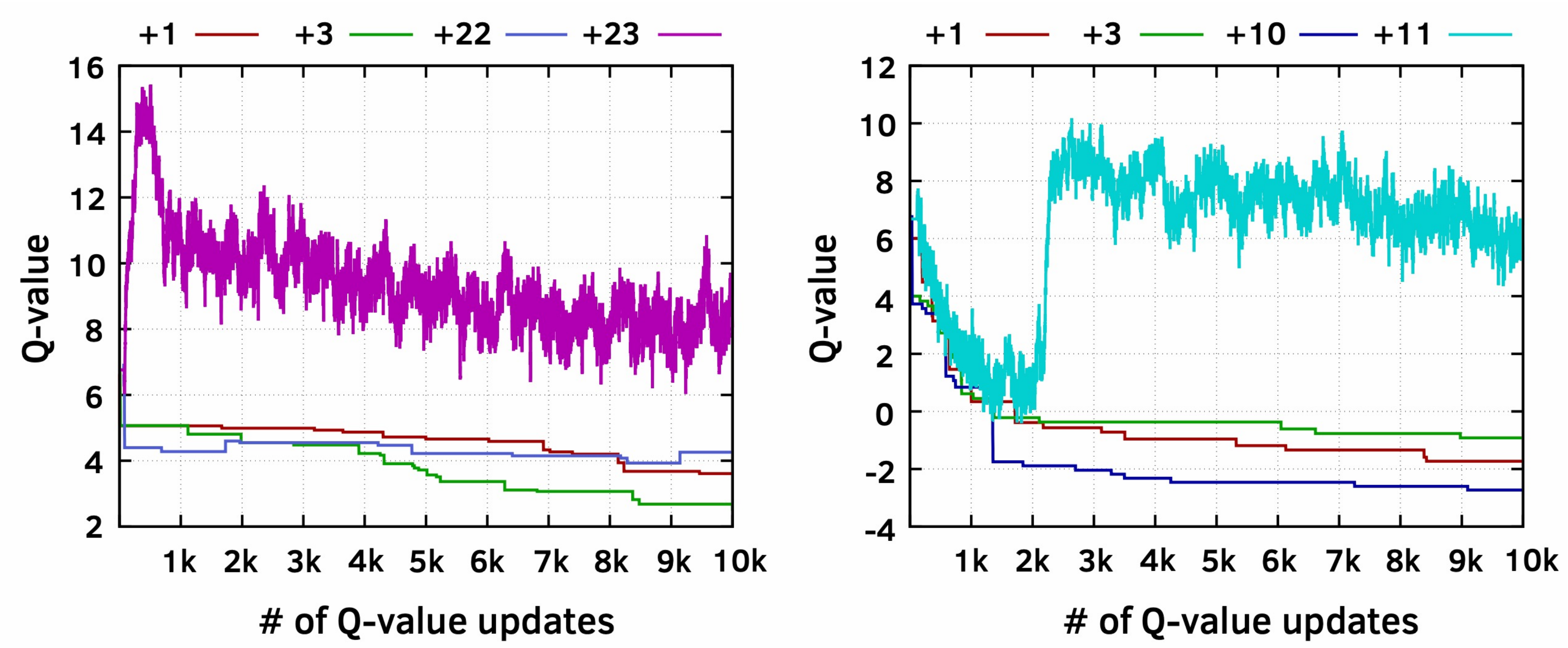


Figure 13: Q-value curves of `PC+Delta` feature values (a) `0x436a81+0` and (b) `0x4377c5+0` in `459.GemsFDTD-1320B`.

Customizing Rewards

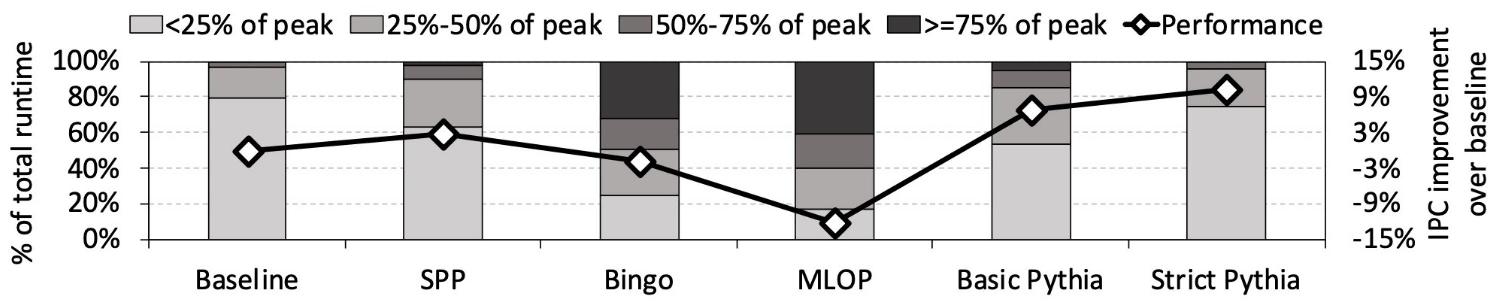


Figure 14: Performance and main memory bandwidth usage of prefetchers in `Ligra-CC`.

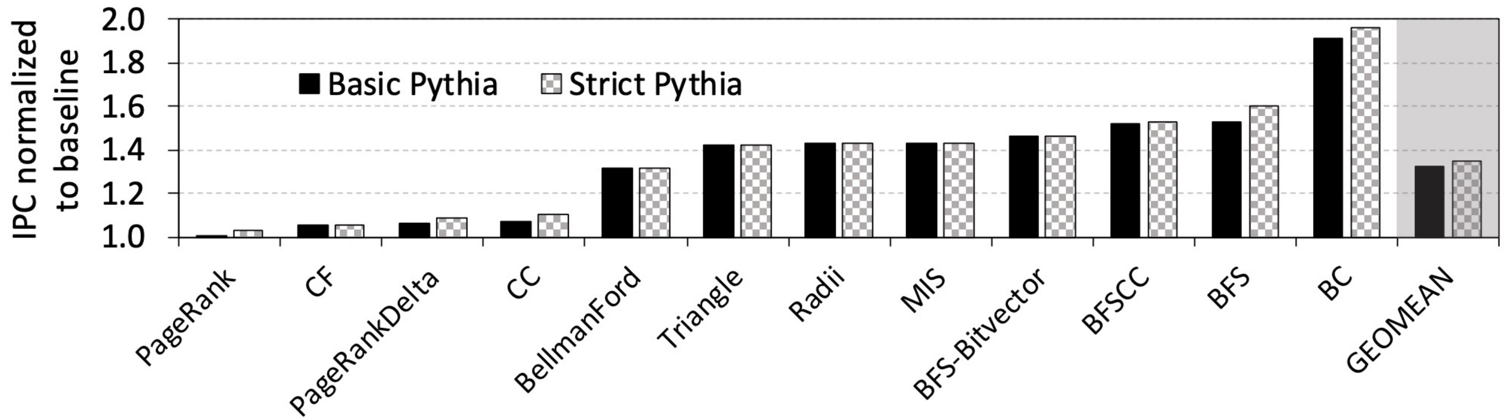


Figure 15: Performance of the basic and strict Pythia configurations on the `Ligra` workload suite.

Customizing Features

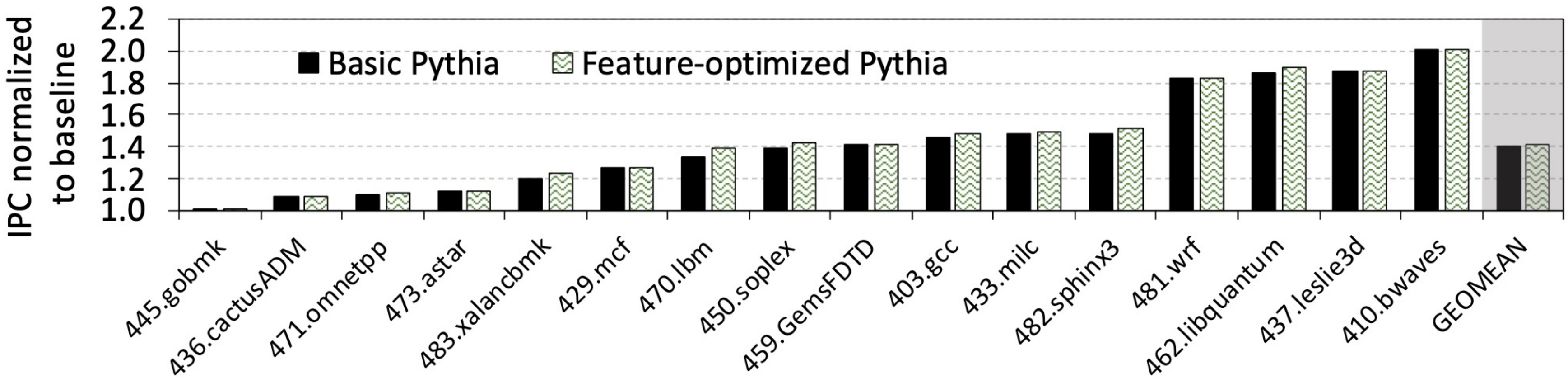


Figure 16: Performance of the basic and feature-optimized Pythia on the `SPEC CPU2006` suite.

BACKUP

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

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

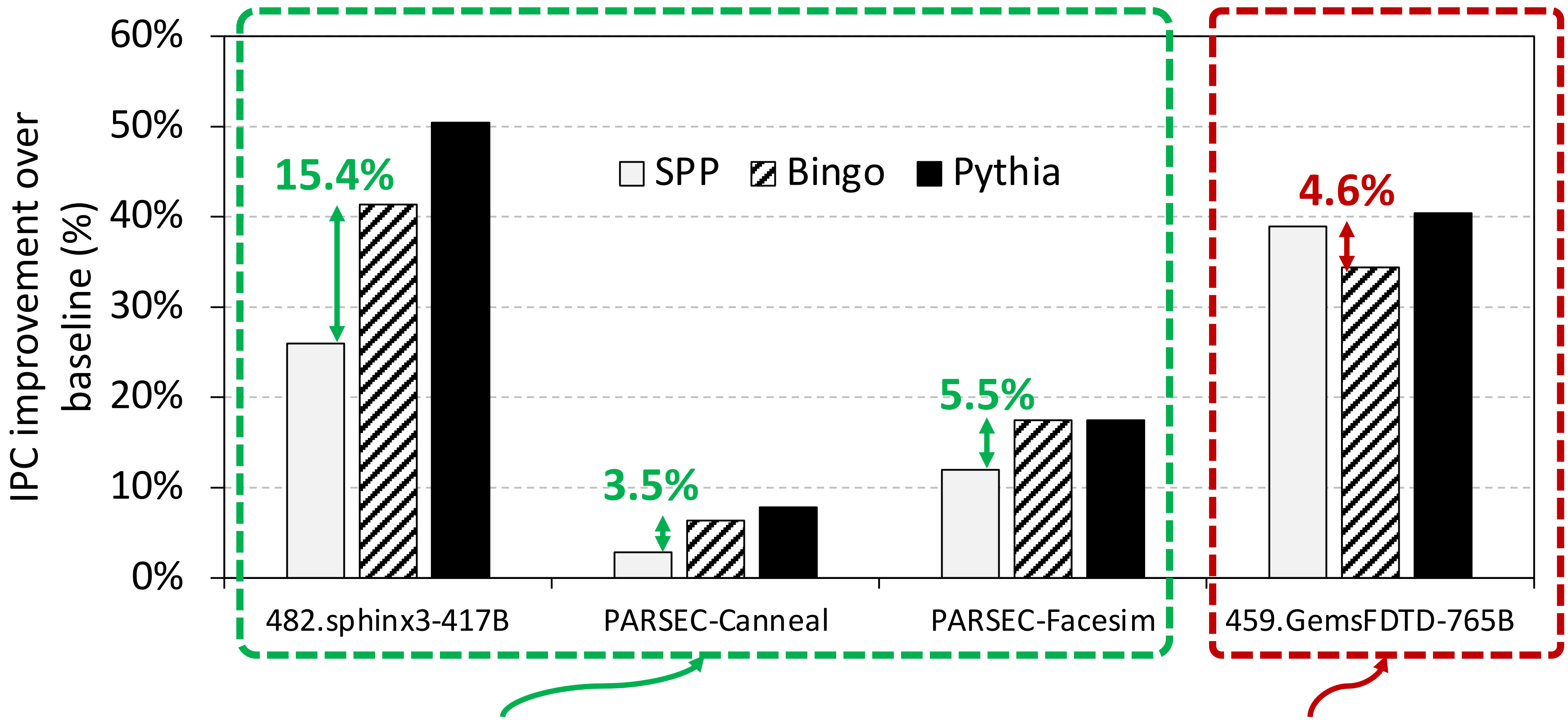
1 Predict mainly using a **single program feature**

2 Lack inherent **system awareness**

3 Lack **in-silicon customizability**

(1) Single-Feature Prefetch Prediction

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



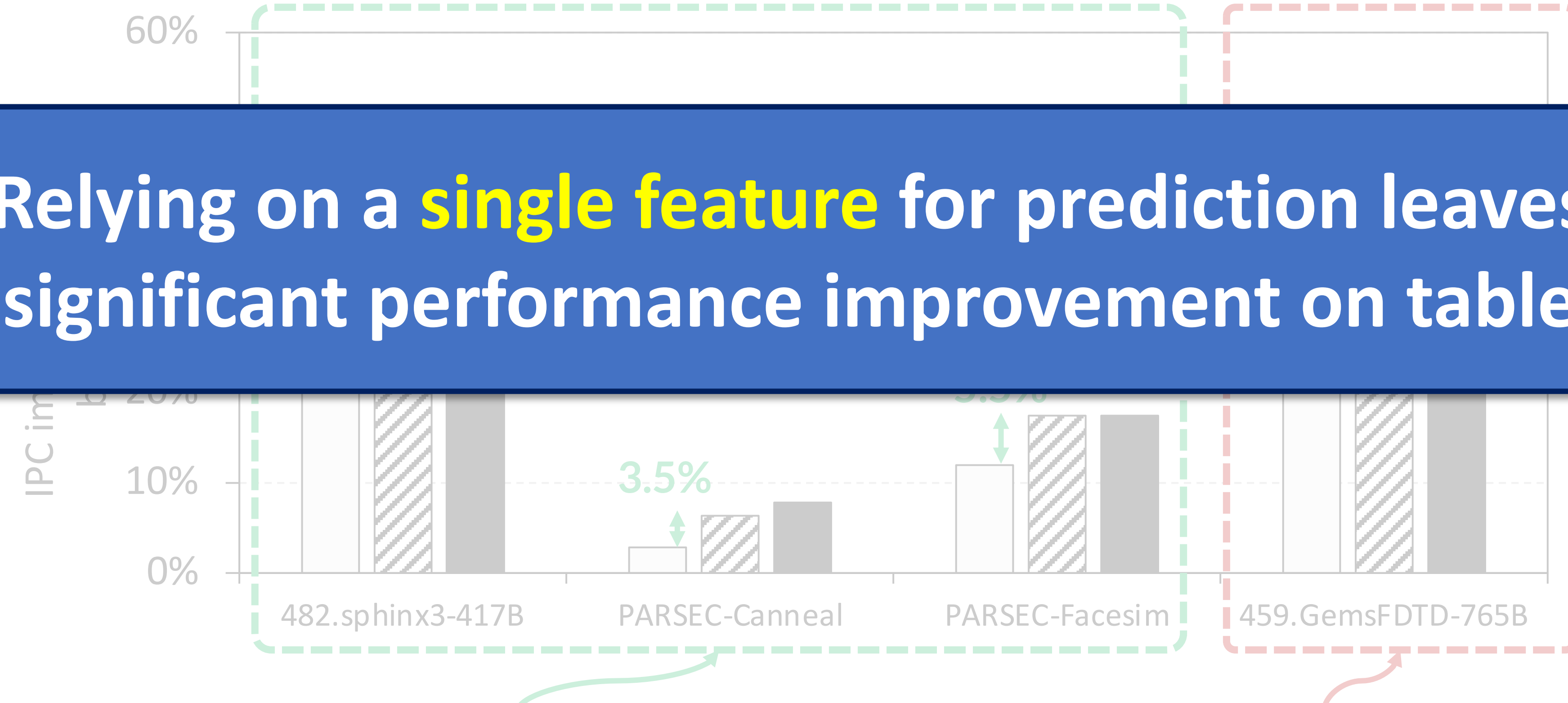
Bingo [1] performs better

SPP [2] performs better

[1] Bakshalipour 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

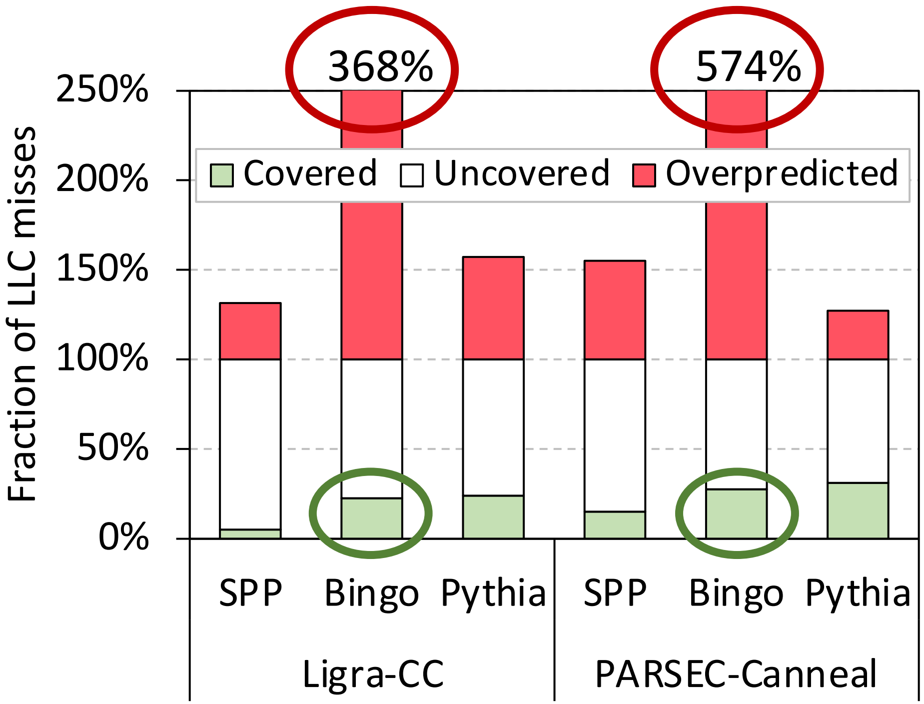
Bingo [1] performs better

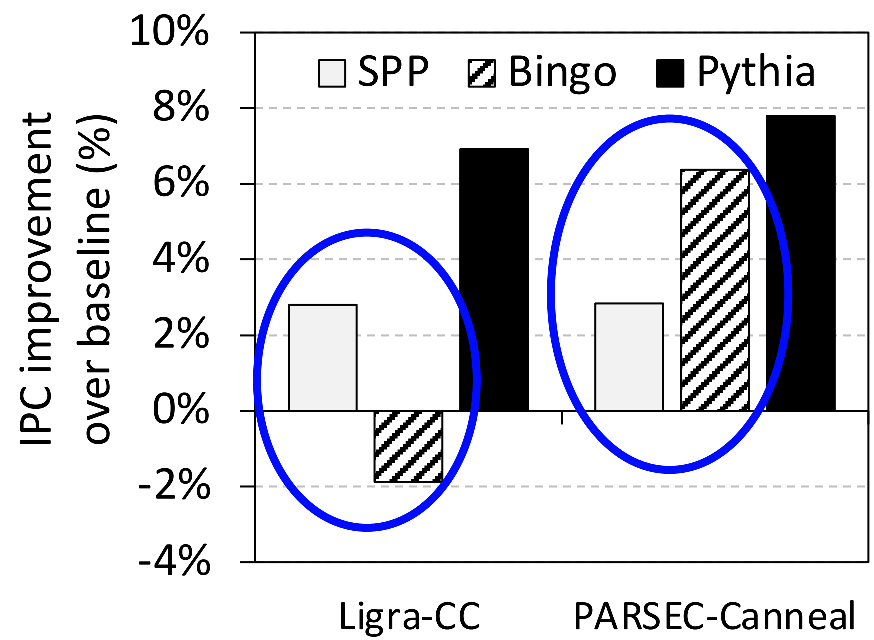
SPP [2] performs better

[1] Bakshalipour et al., HPCA'19 [2] Kim et al., MICRO'16

(2) Lack of Inherent System Awareness

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

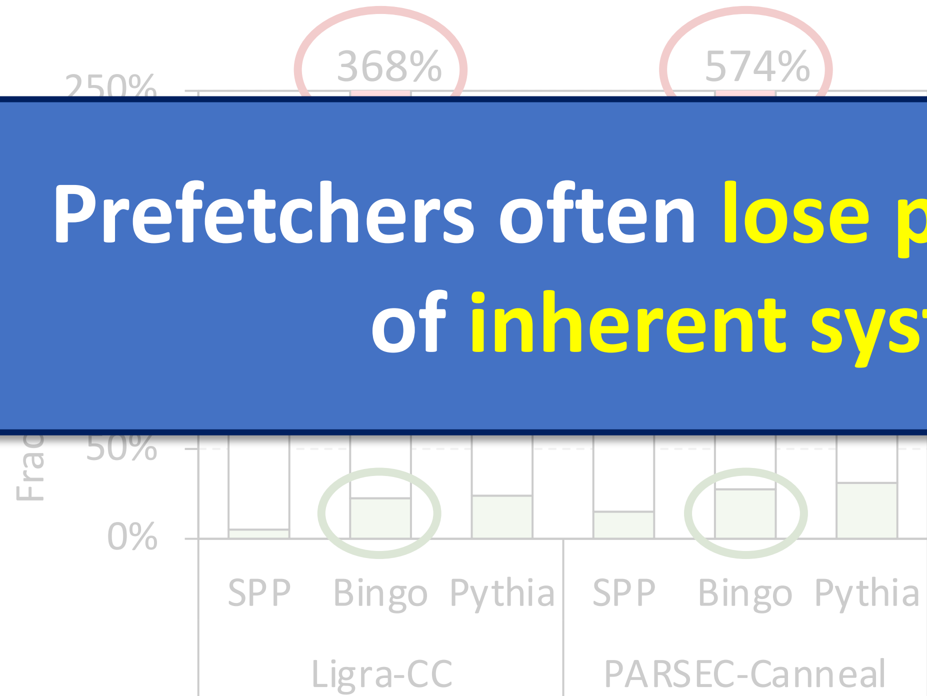


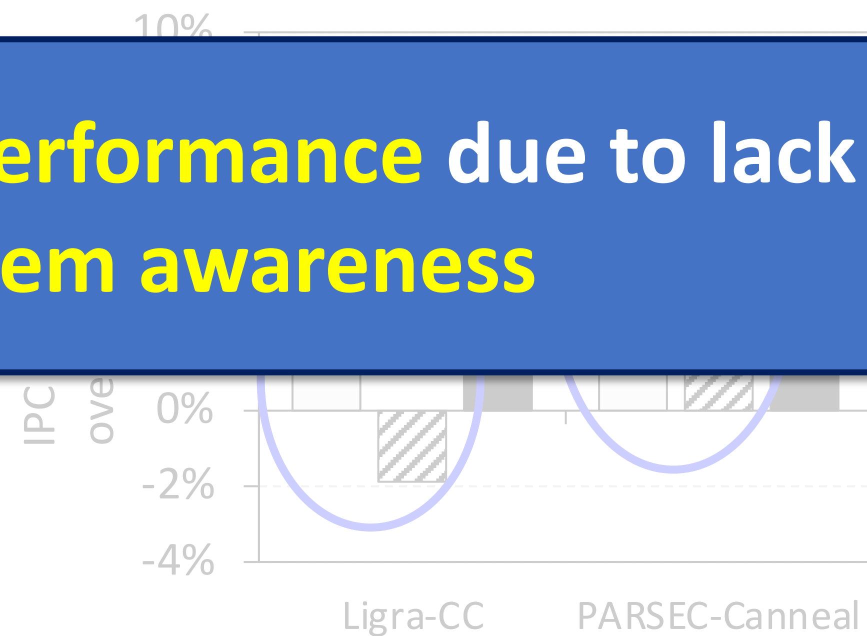


Similar coverage **Lower overpredictions** **Yet, lower performance**

(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 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

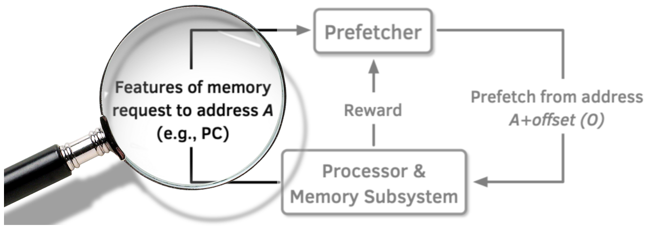
Formulating Prefetching as RL

What is State?

- ***k*-dimensional** vector of features

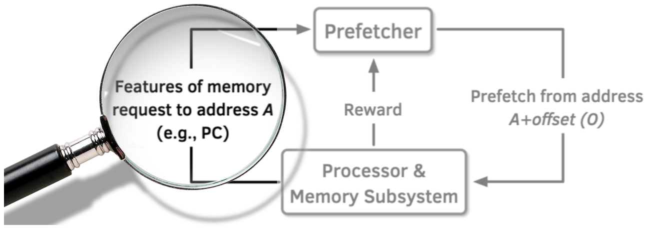
$$S \equiv \{\phi_S^1, \phi_S^2, \ldots, \phi_S^k\}$$

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

Feature-1 (ϕ_1)

Feature-2 (ϕ_2)

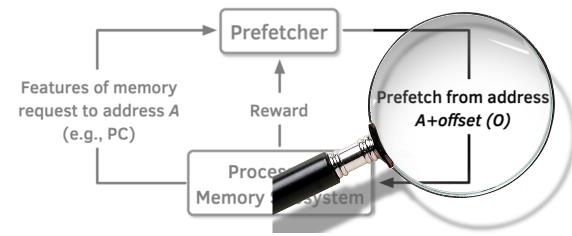
PC
(Control-flow info.)

Cacheline Delta
(Data-flow info.)

Seq. of last-4 deltas
(Data-flow info.)

What is Action?

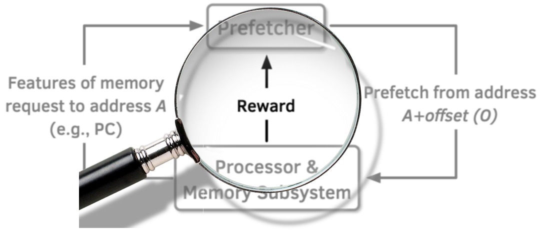
Given a demand access to address A
the action is to **select prefetch offset "O"**



- **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?

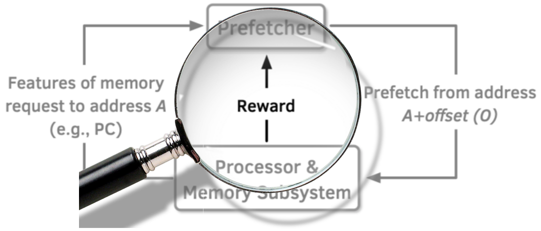
- Defines the **objective** of Pythia



- 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?

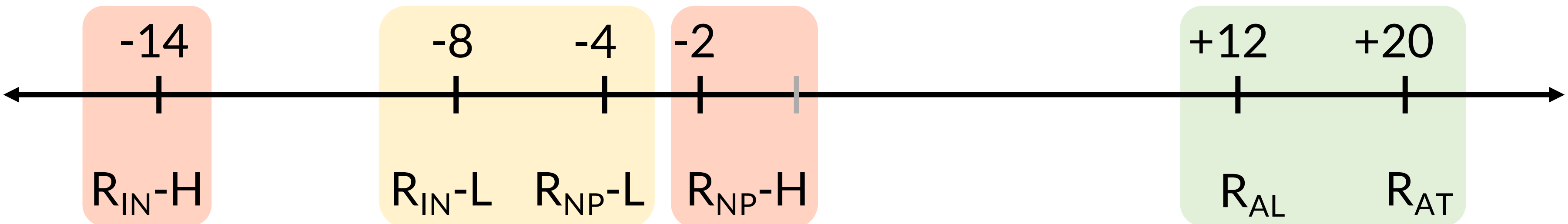


- **Seven** distinct reward levels
  - *Accurate and timely* (R_{AT})
  - *Accurate but late* (R_{AL})
  - *Loss of coverage* (R_{CL})
  - *Inaccurate*
    - With low memory b/w usage (R_{IN}-L)
    - With high memory b/w usage (R_{IN}-H)
  - *No-prefetch*
    - With low memory b/w usage (R_{NP}-L)
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



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

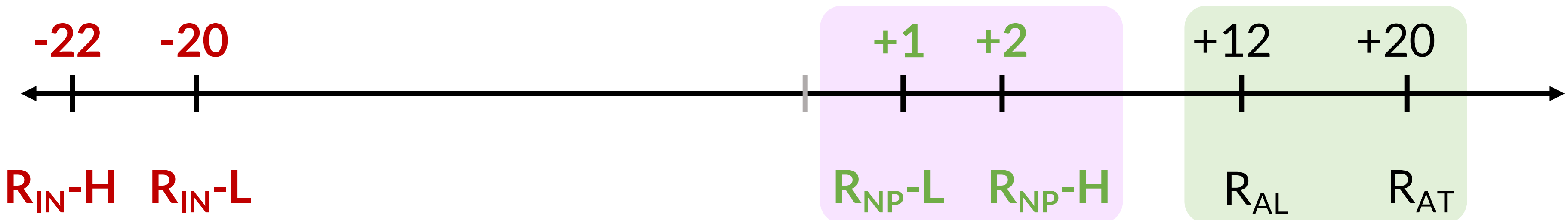
1 **Highly prefers to generate accurate prefetches**

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

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

Steering Pythia's Objective via Reward Values

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



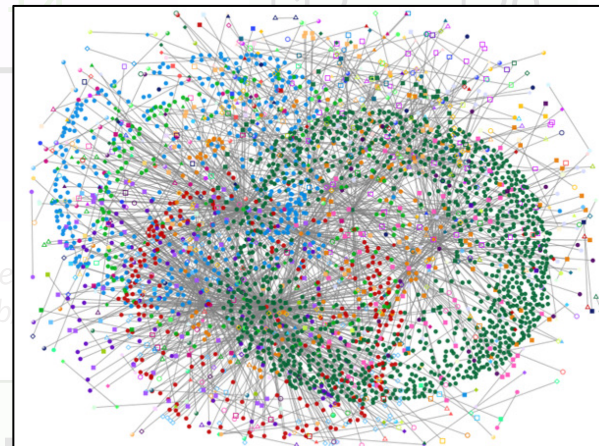
AT = *Accurate & timely*; **AL** = *Accurate & late*; **NP** = *No-prefetching*; **IN** = *Inaccurate*;
H = *High mem. b/w*; **L** = *Low mem. b/w*

1 **Highly prefers to generate accurate prefetches**

2 **Otherwise prefers not to prefetch**

Steering Pythia's Objective via Reward Values

Strict Pythia configuration



Server-class processors

Bandwidth-sensitive workloads

Talk Outline

Key Shortcomings of Prior Prefetchers

Formulating Prefetching as Reinforcement Learning

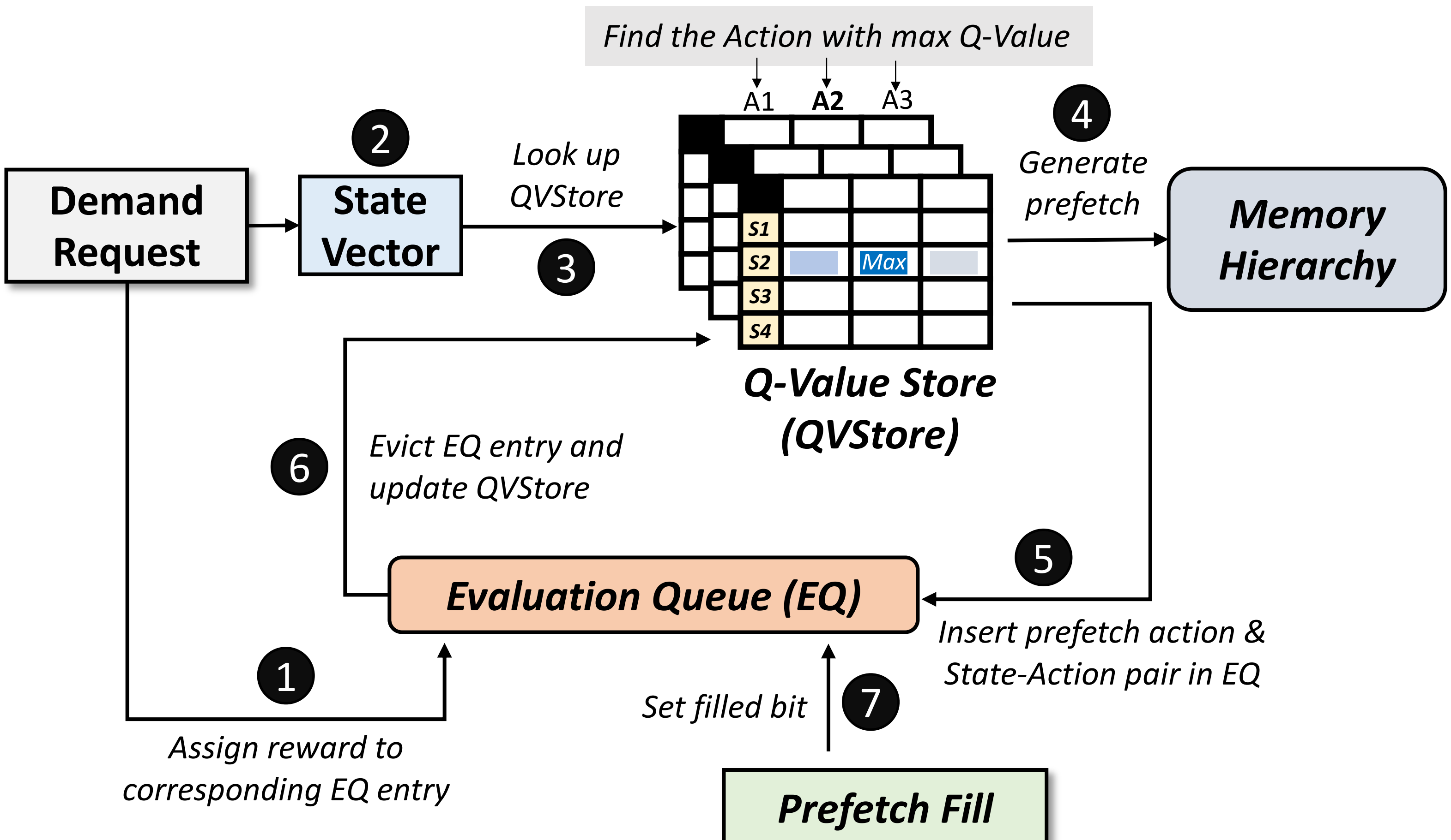
Pythia: Overview

Evaluation of Pythia and Key Results

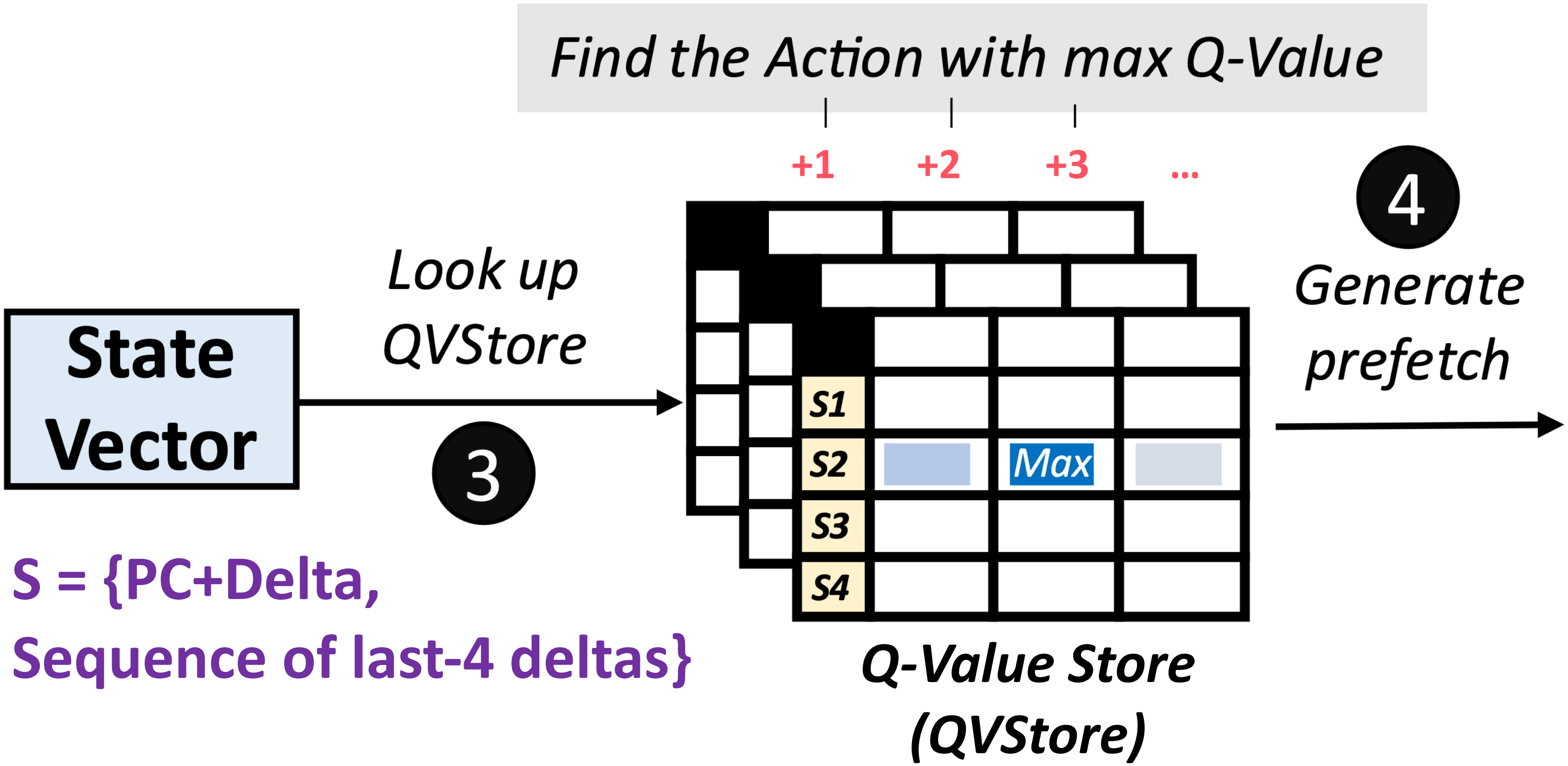
Conclusion

Pythia Overview

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



Architecting QVStore



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

Architecting QVStore

Fast prefetch prediction

Fast retrieval of Q-values from QVStore

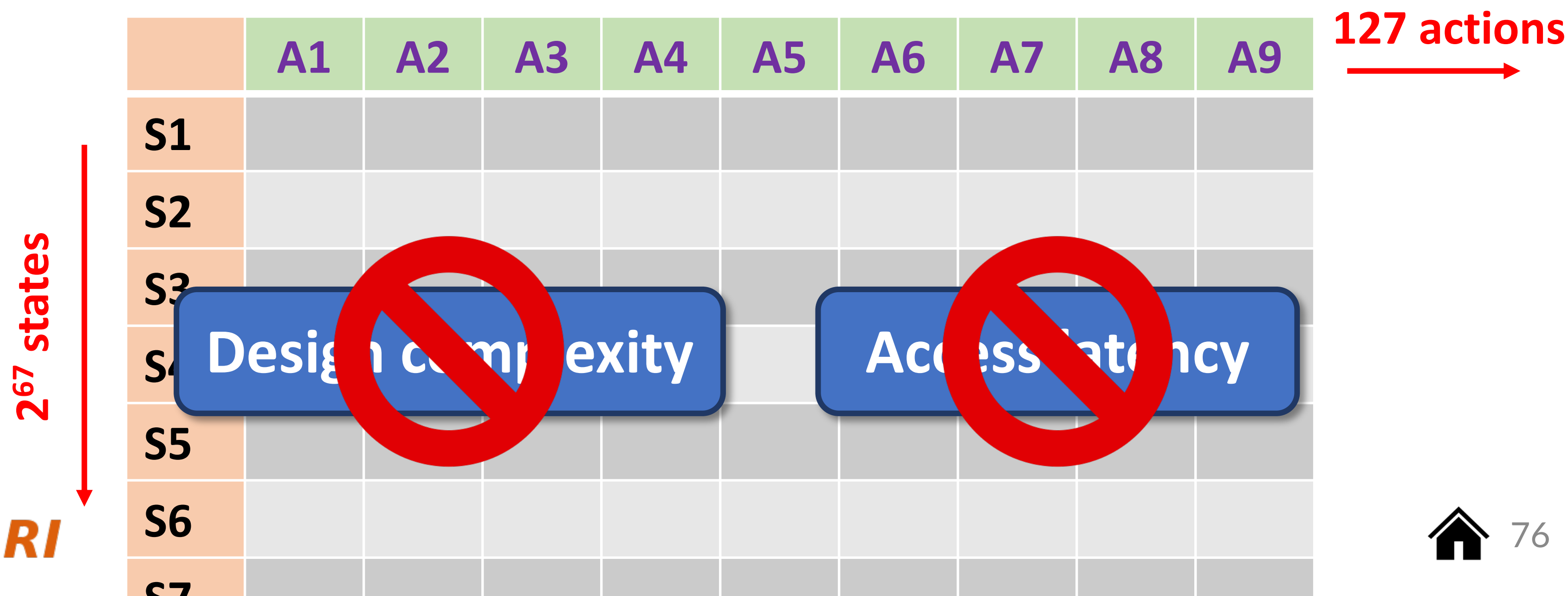
Efficient storage organization of Q-values in QVStore

Organization of QVStore

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

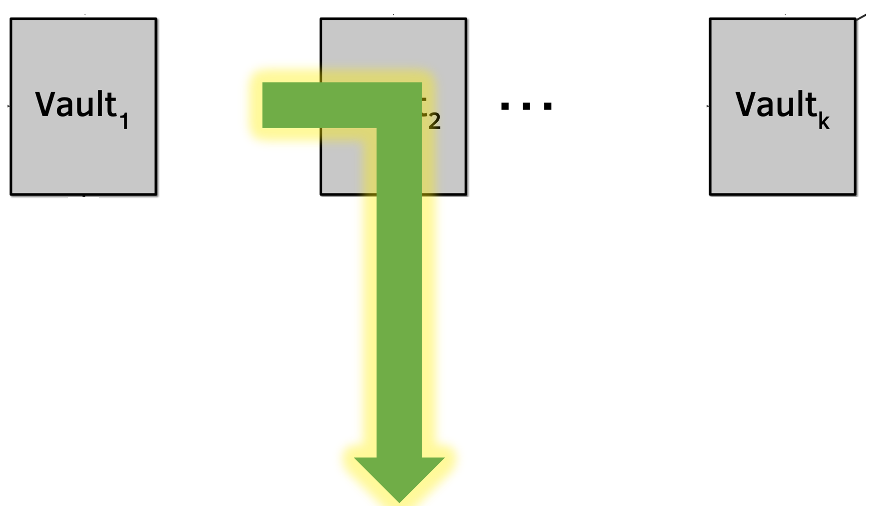
S = {PC+Delta, Sequence of last-4 deltas}

32b + 7b + 4x7b = 67 bits



Organization of QVStore

- We partition QVStore into ***k* vaults** [*k* = 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 **MAX** of all feature-action Q-values

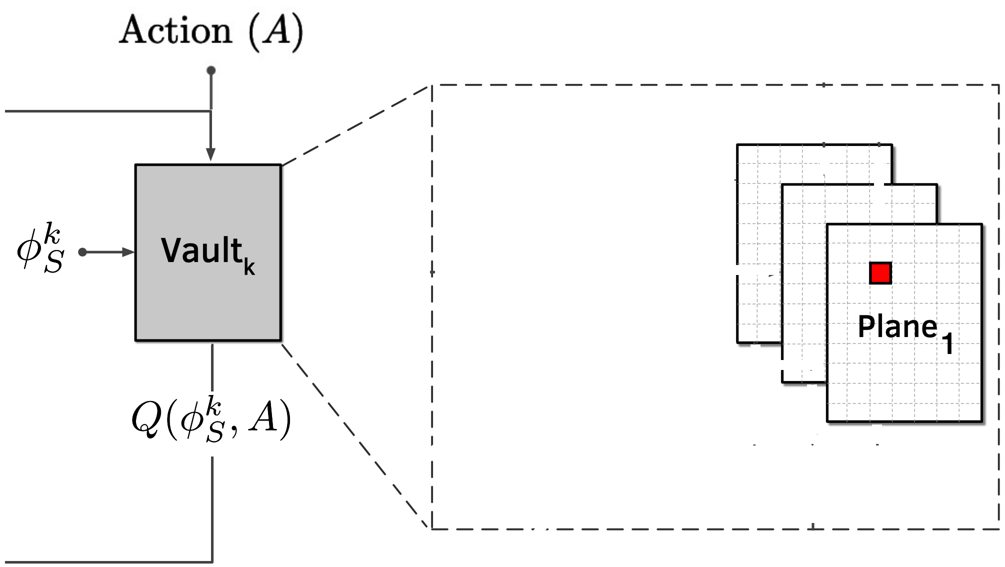
MAX ensures the Q(S,A) is driven by the constituent feature that has highest Q(ϕ,A)

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(ϕ,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 parital 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

- Query **each plane in parallel** with hashed feature and action

2. Reduces chances of sharing partial Q-values across widely different feature values

- Compute **SUM** of all partial feature-action Q-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

- Reward a

https://arxiv.org/pdf/2109.12021.pdf

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]

[3] https://github.com/ChampSim/ChampSim

Basic Pythia Configuration

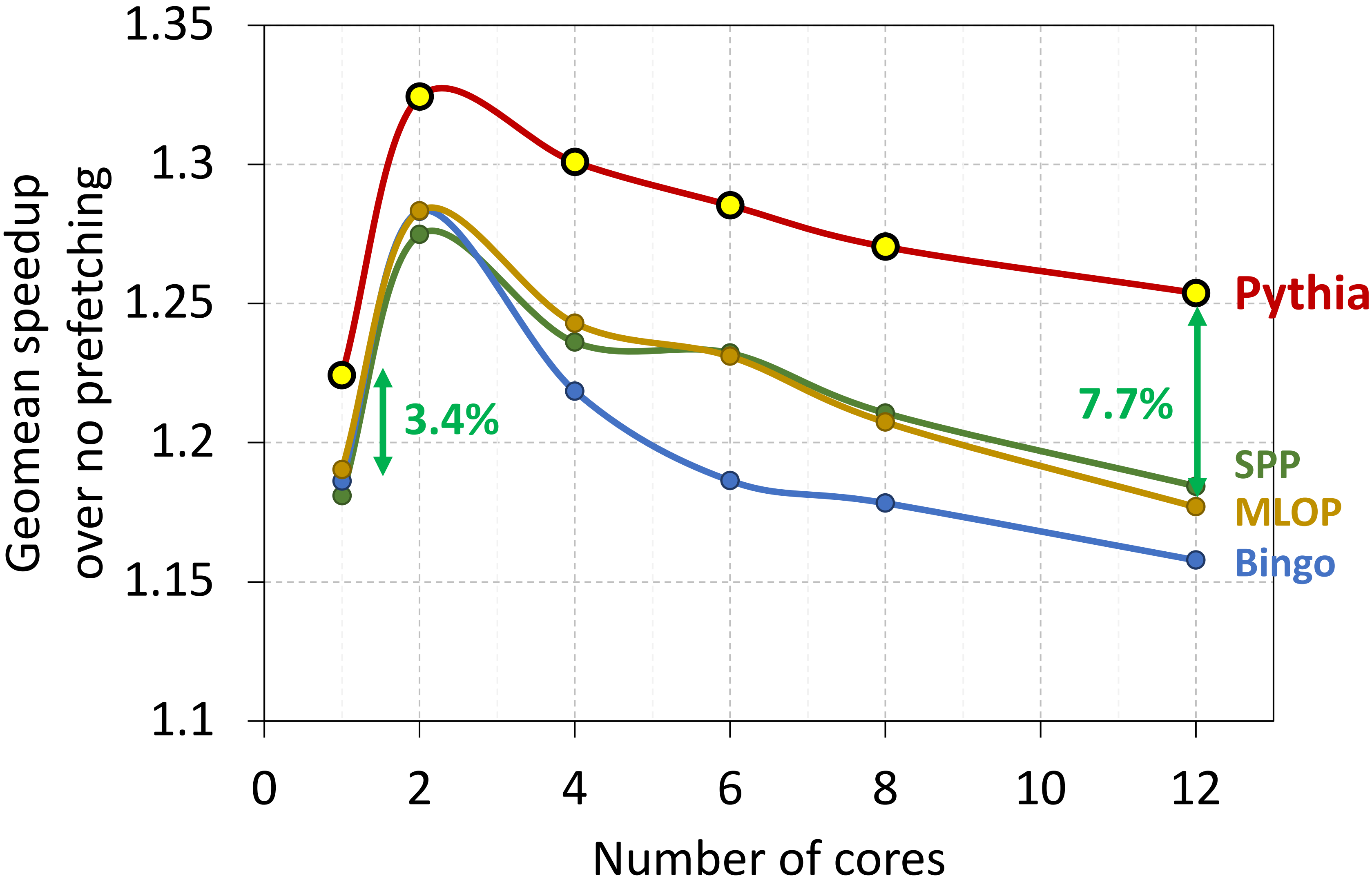
- Derived from **automatic design-space exploration**

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

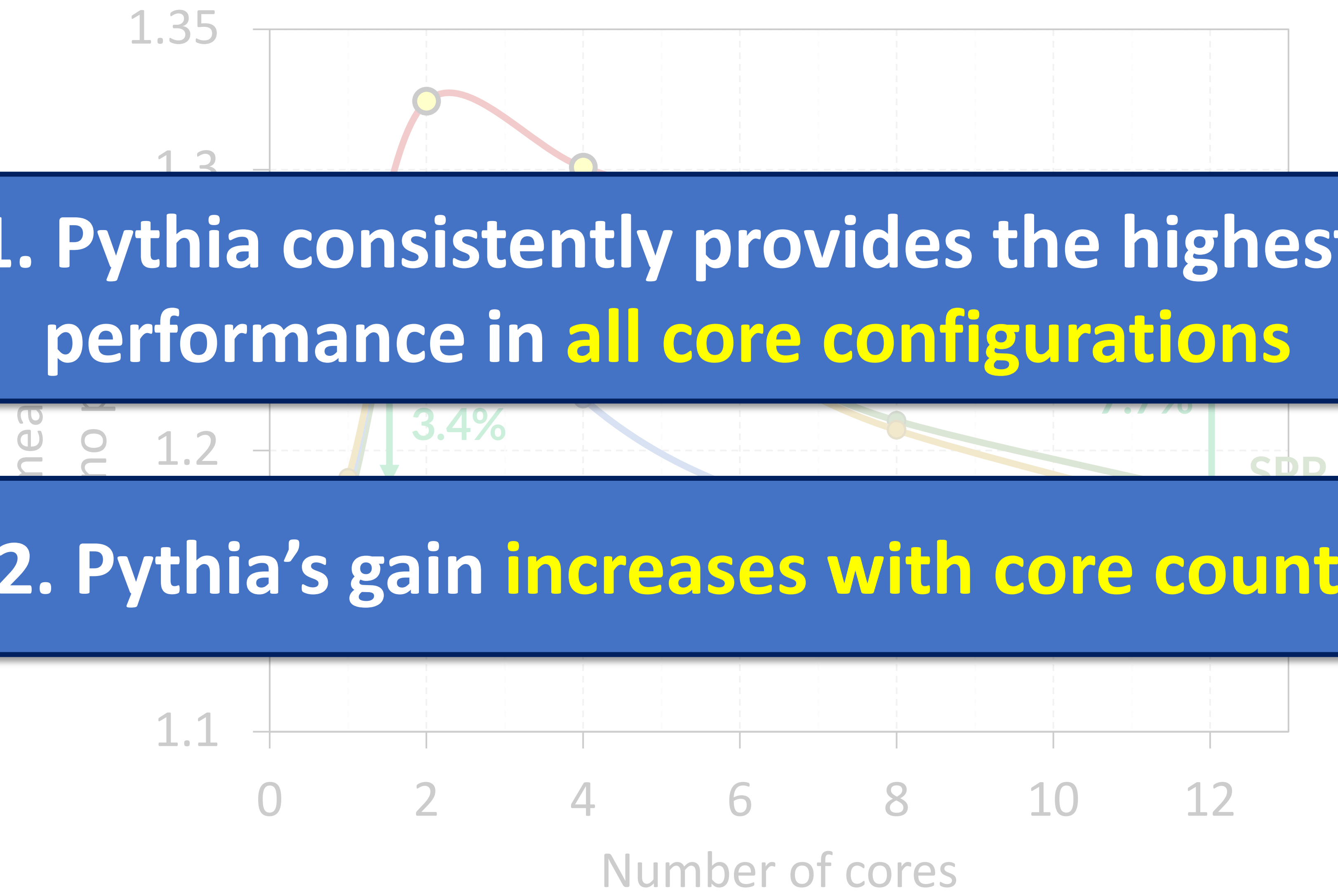
- **Actions:** 16 prefetch offsets
  - Ranging between -6 to +32. Including 0.

- **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

Performance with Varying Core Count



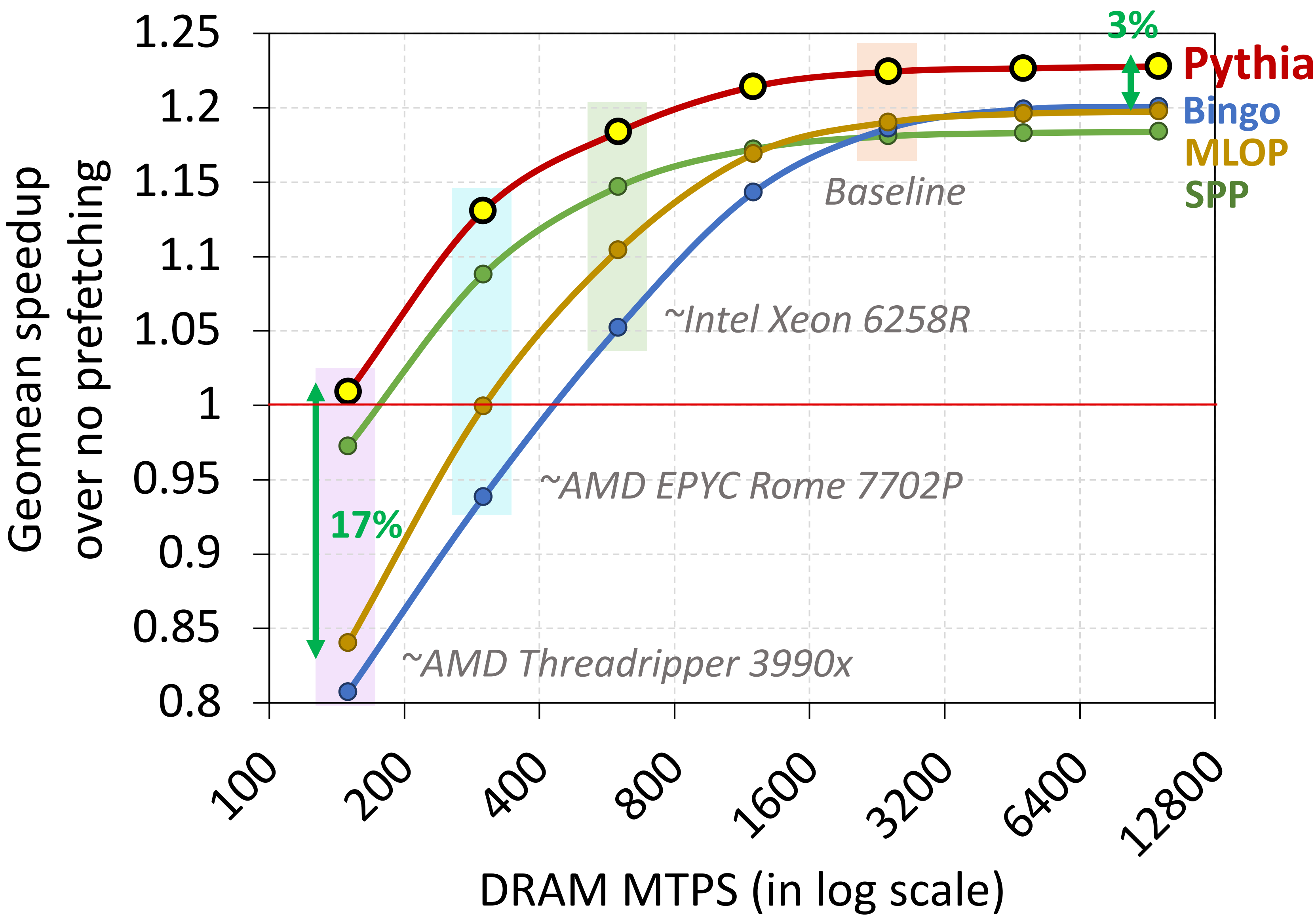
Performance with Varying Core Count



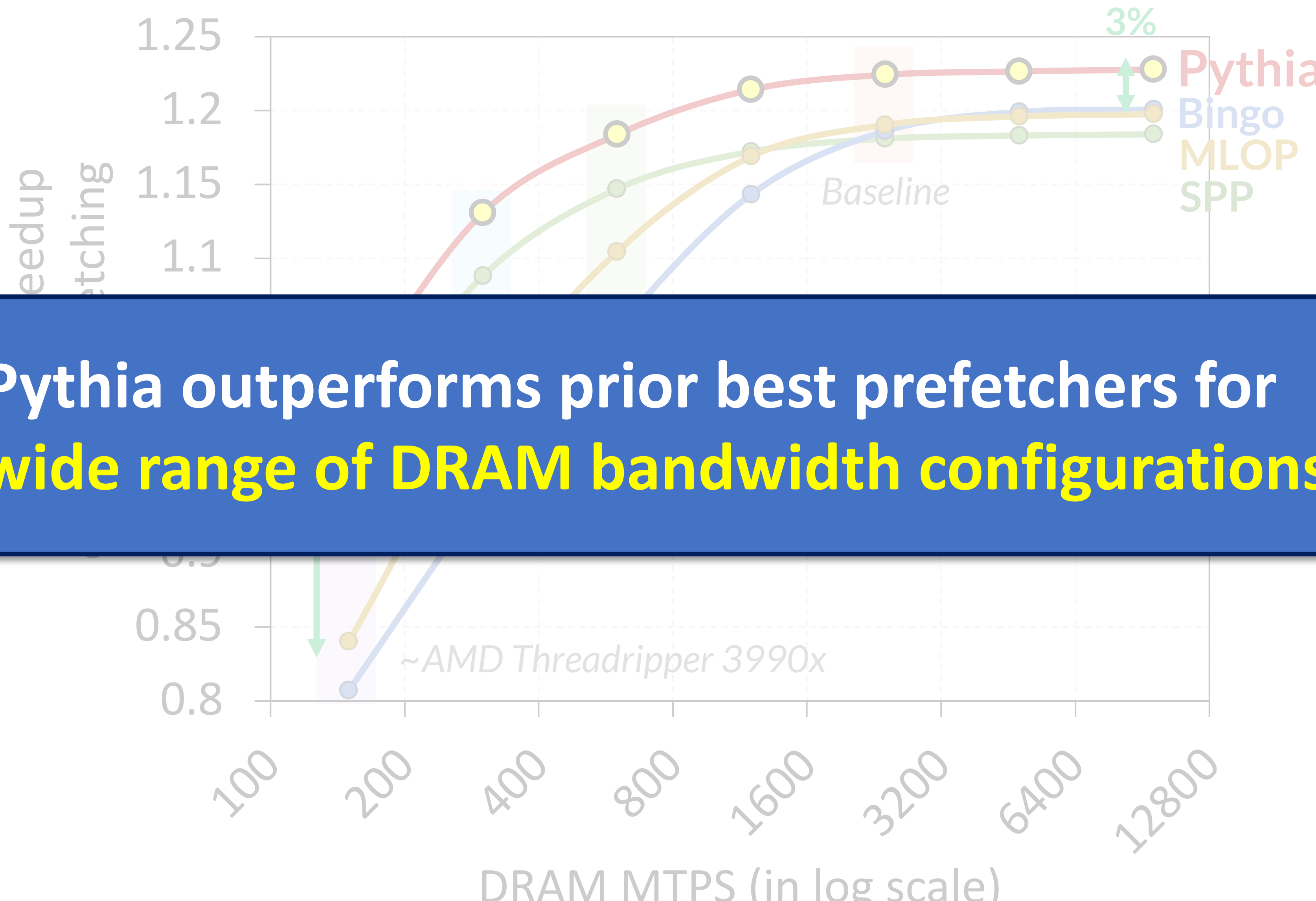
1. Pythia consistently provides the highest performance in **all core configurations**

2. Pythia's gain **increases with core count**

Performance with Varying DRAM Bandwidth



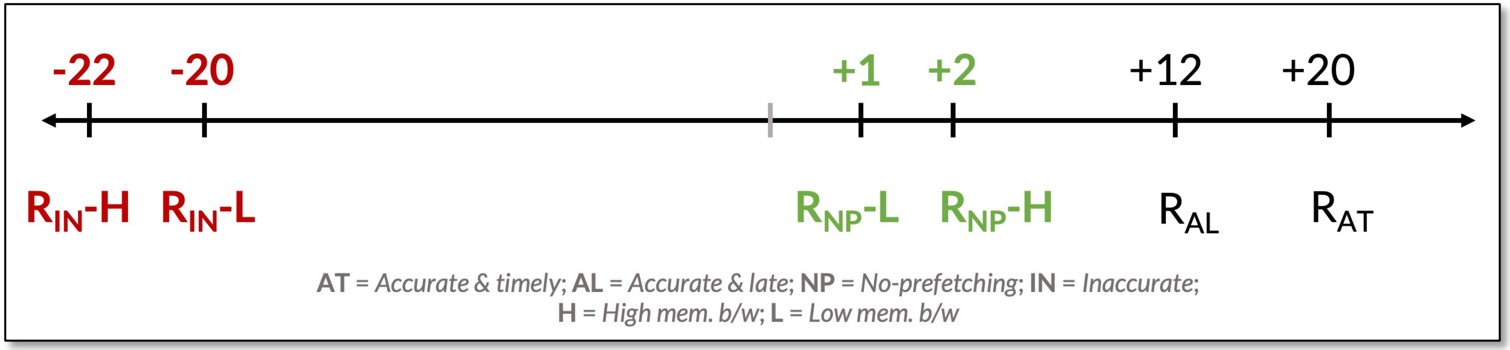
Performance with Varying DRAM Bandwidth



Pythia outperforms prior best prefetchers for
a wide range of DRAM bandwidth configurations

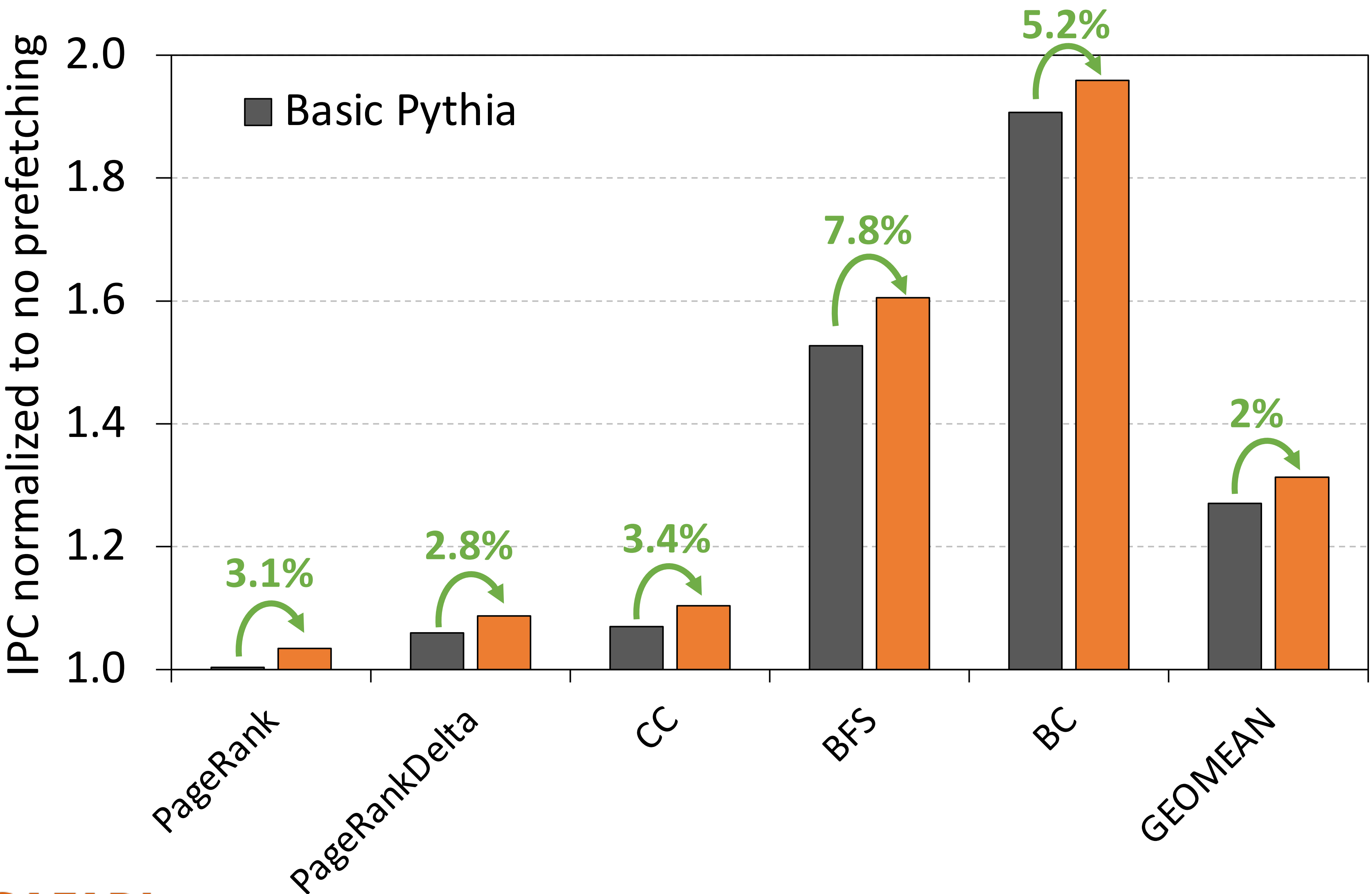
Performance Improvement via Customization

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

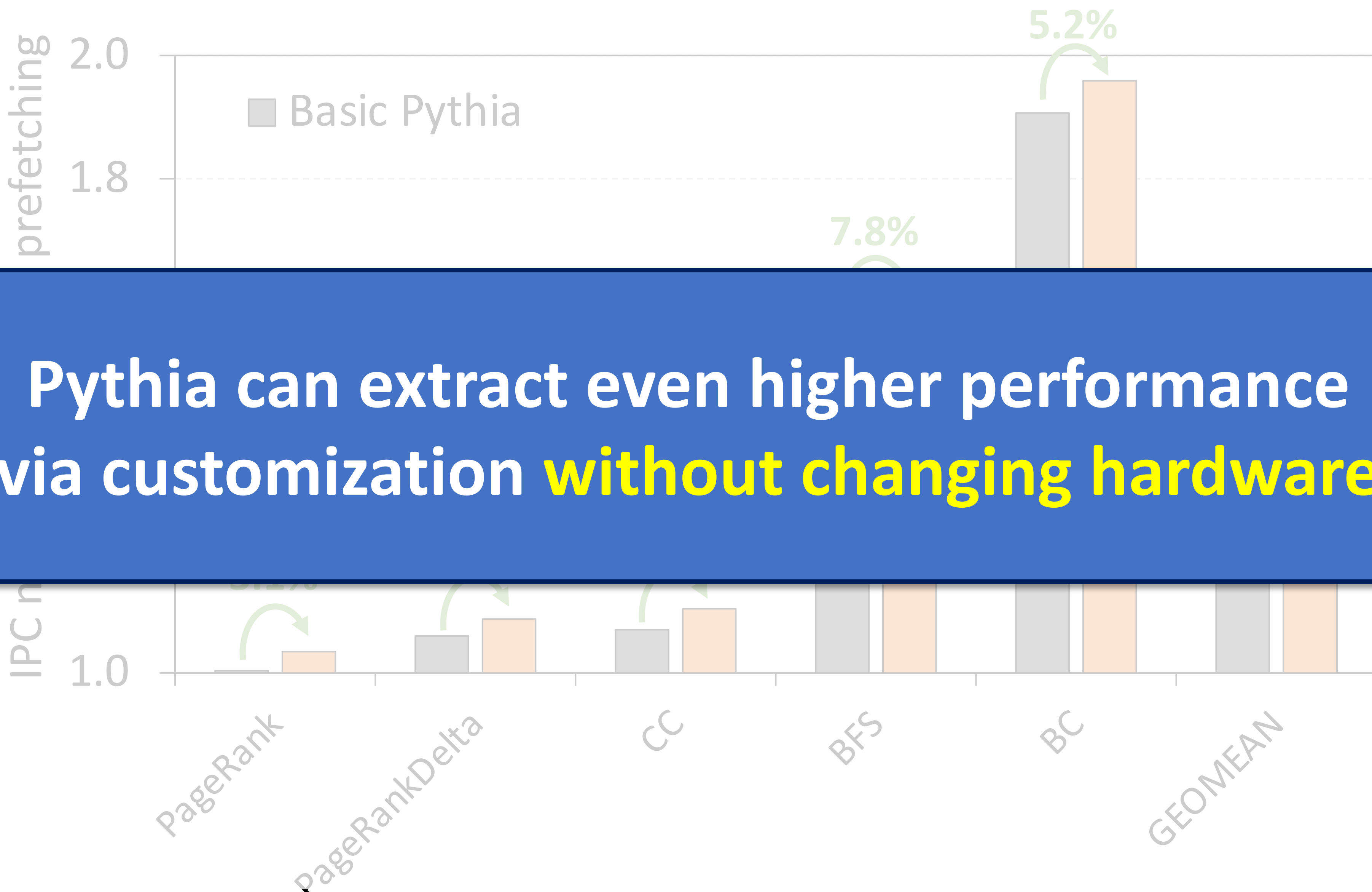


- Strict Pythia is **more conservative** in generating prefetch requests than the basic Pythia
- Evaluate on all **Ligra graph processing workloads**

Performance Improvement via Customization



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)

[4] https://www.chisel-lang.org

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: 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

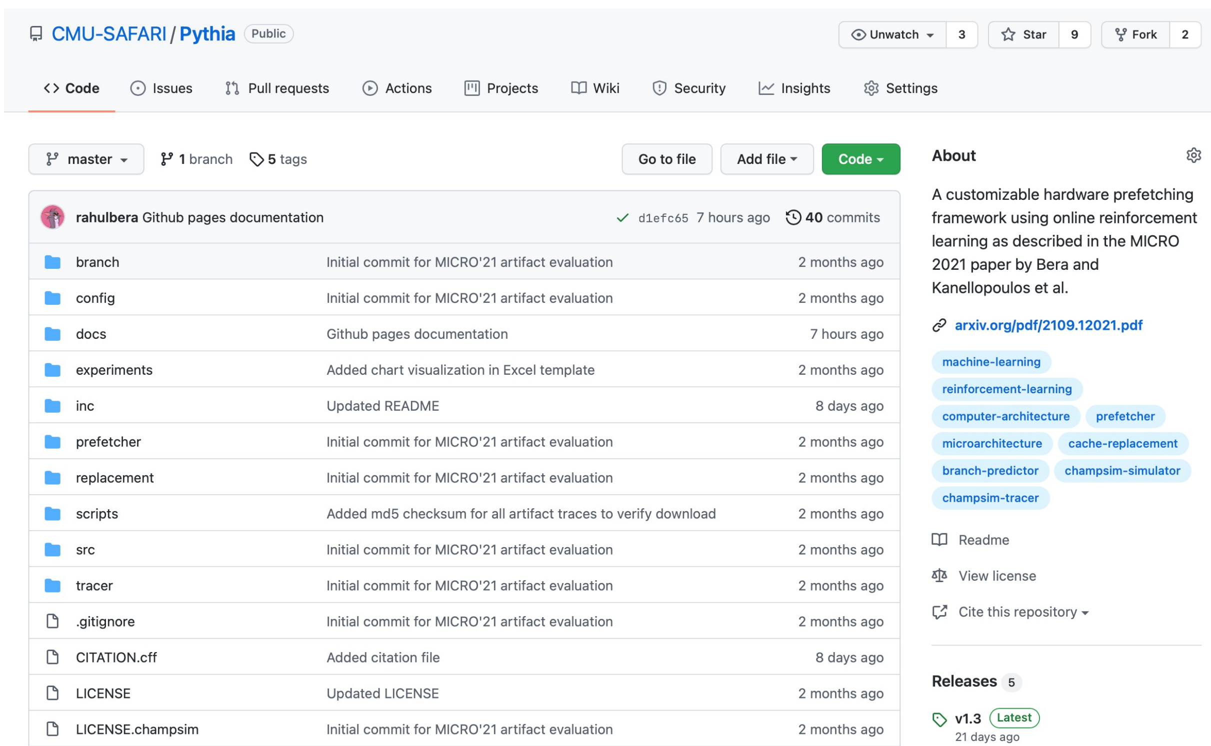
https://arxiv.org/pdf/2109.12021.pdf

- Performance sensitivity towards different features and hyperparameter values
- Detailed single-core and four-core performance

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



Talk Outline

Key Shortcomings of Prior Prefetchers

Formulating Prefetching as Reinforcement Learning

Pythia: Overview

Evaluation of Pythia and Key Results

Conclusion

Executive Summary

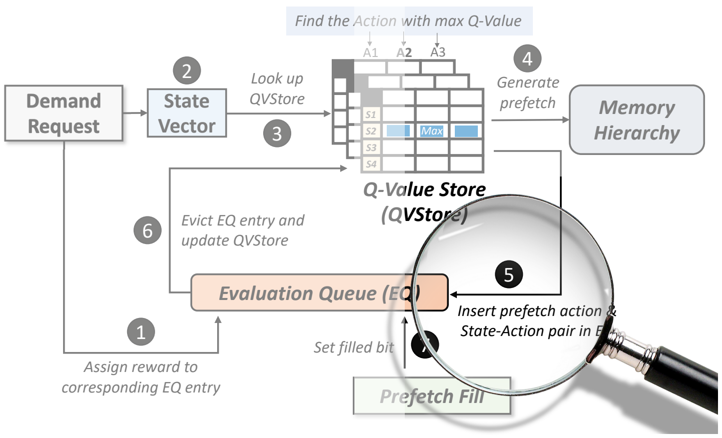
- **Background**: Prefetchers predict addresses of future memory requests by associating memory access patterns with program context (called **feature**)
- **Problem**: Three key shortcomings of prior prefetchers:
  - Predict mainly using a **single program feature**
  - Lack **inherent system awareness** (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

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

Reward Assignment to EQ Entry

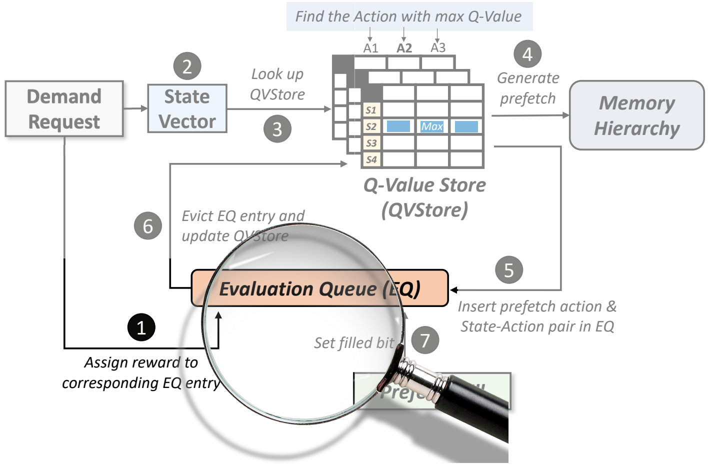
- **Every** action gets inserted into EQ
- Reward is assigned to each EQ entry **before or during** the eviction

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



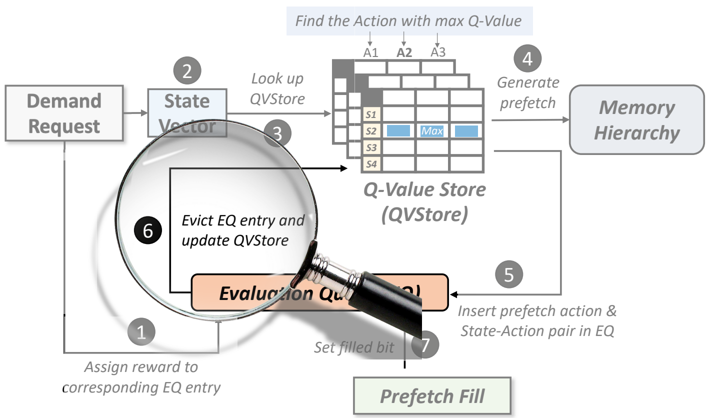
Reward Assignment to EQ Entry

- **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*)