Pythia

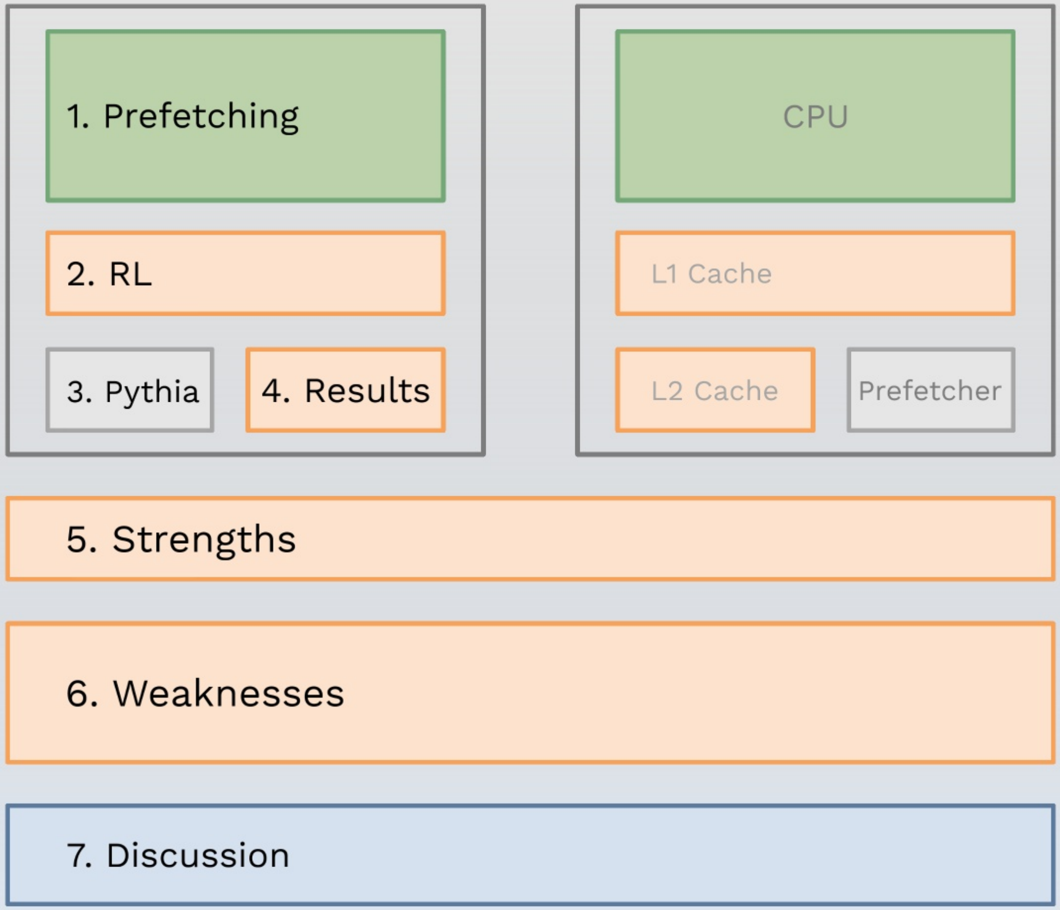
A Customizable
Hardware Prefetching Framework
Using Online Reinforcement Learning

Presented by
Cedric Caspar

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

ETH Zürich, Processor Architectur Research Labs, Intel Labs, TU Delft

Overview


1. Prefetching
2. RL
3. Pythia
4. Results
CPU
L1 Cache
L2 Cache
Prefetcher
5. Strengths
6. Weaknesses
7. Discussion

Executive Summary

Background: Prefetchers predict address of future memory requests by finding access patterns from program context / feature

Problem: Three key shortcomings of prior prefetchers:

- Using only single program feature
- Lack of system awareness / feedback
- Lack of in-silicon customizability

Goal: Design adaptive and multi-feature prefetching framework

Contribution: Pythia, formulating prefetching as a reinforcement learning problem

Results:

- Evaluated using wide range of workloads
- Outperforms current best prefetchers by 3.4%, 7.7% & 17% in 1/4/bw-constrained cores

1

1. Prefetching

2

1. Prefetching

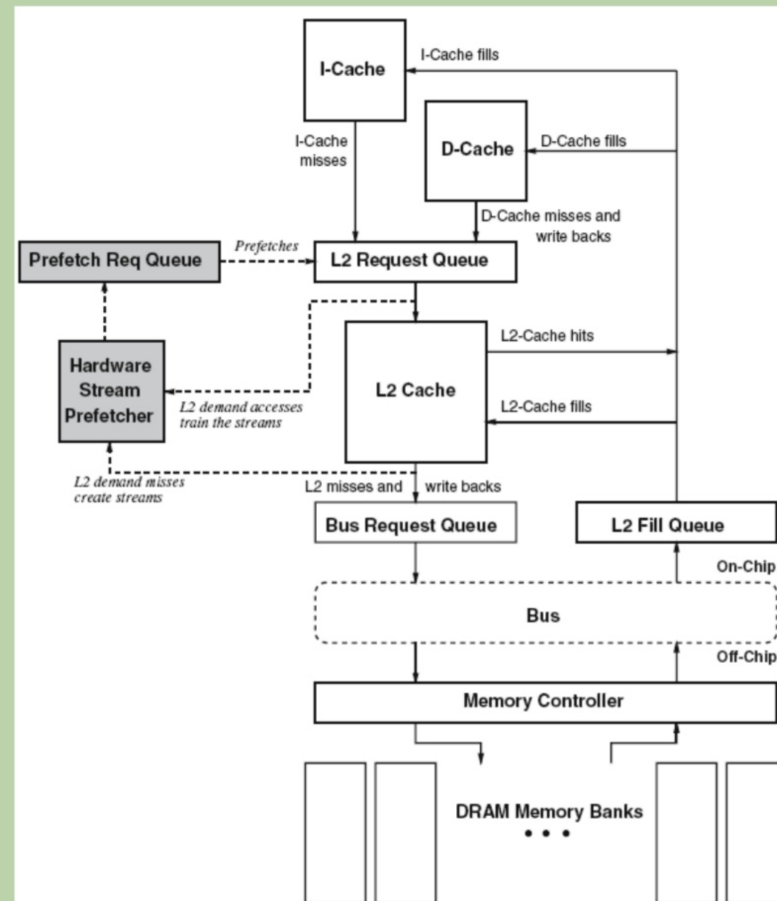
- **DRAM latency** remains a critical bottleneck
- **Spatial locality** provides significant performance benefits
- **Irregular patterns** are difficult, inaccurate, hardware intensive

2

1. Prefetching

- **DRAM latency** remains a critical bottleneck
- **Spatial locality** provides significant performance benefits
- **Irregular patterns** are difficult, inaccurate, hardware intensive

- **Solutions**:
  - Reduce latency
  - Tolerate latency via multithreading
  - Hide latency via caching/prefetching



2

Current Prefetchers

3

Current Prefetchers

- **Simplest**: Next-Line or Stride prefetcher

3

Current Prefetchers

- **Simplest**: Next-Line or Stride prefetcher
- Cache-block address based stride prefetcher
  => **Stream buffer** prefetcher

Hardware intensive

3

Current Prefetchers

- **Simplest**: Next-Line or Stride prefetcher
- Cache-block address based stride prefetcher
  => **Stream buffer** prefetcher — Hardware intensive
- **Locality** based prefetching — Bandwidth intensive

3

Current Prefetchers

- **Simplest**: Next-Line or Stride prefetcher
- Cache-block address based stride prefetcher
  => **Stream buffer** prefetcher

Hardware intensive

- **Locality** based prefetching

Bandwidth intensive

Key Ideas for Pythia:
- **Adaptive** to access pattern switch
- Memory **bandwidth consideration**
- **Parametric** variability for the prefetcher

3

2. Reinforcement Learning

4

2. Reinforcement Learning

- Different **form** of **Machine Learning**

4

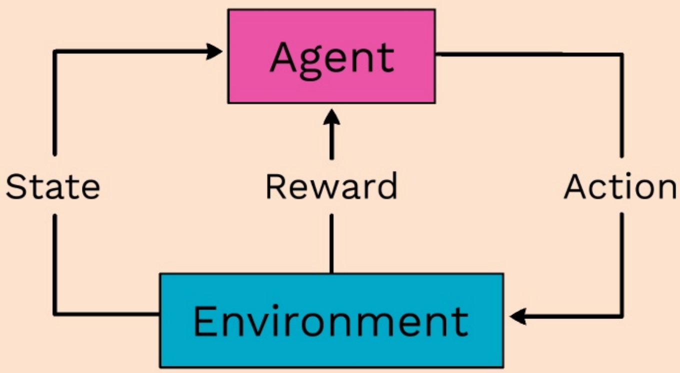
2. Reinforcement Learning

- Different **form** of **Machine Learning**
- **No training data** needed in advance (online)

4

2. Reinforcement Learning

- Different **form** of **Machine Learning**
- **No training data** needed in advance (online)

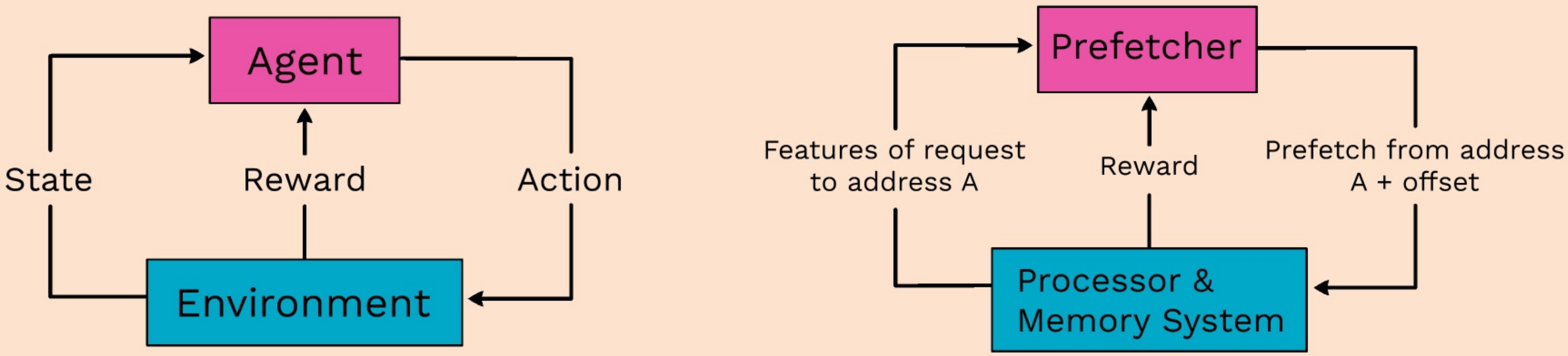


- **Q-Values** for each state-action pair represent **expected reward**

4

2. Reinforcement Learning

- Different **form** of **Machine Learning**
- **No training data** needed in advance (online)



- **Q-Values** for each state-action pair represent **expected reward**

4

What is State?

5

What is State?

k-dimensional feature vector

feature = **control flow** component + **data flow** component

	control flow component	data flow component
e.g.	- PC	- Cacheline Address
	- Branch PC	- Physical Page Number
	- Last 3 PCs, ...	- last 4 deltas, ...

5

What is State?

k-dimensional feature vector

feature	=	**control flow** component	+	**data flow** component
e.g.		- PC - Branch PC - Last 3 PCs, ...		- Cacheline Address - Physical Page Number - last 4 deltas, ...

What is Action?

Given a demand address A **select prefetch offset O**

Action range: [-63,63], will be pruned for efficiency

If **zero-offset** selected, **no prefetch** is generated

5

What is Reward?

Defines the **objective** of Pythia encapsulating two metrics:

- **Prefetch usefulness**
- **System feadback**

6

What is Reward?

Defines the **objective** of Pythia encapsulating two metrics:

- **Prefetch usefulness**
- **System feadback**

Many different kinds of rewards get awarded:

Accurate + **timely** (**R**at) | **Accurate** + **late** (**R**al) | **Out of physical page** (**R**cl)
No-prefetch + low/high mem b/w (**R**np-L / **R**np-H)
Inaccurate + low/high mem b/w (**R**in-L / **R**in-H)

6

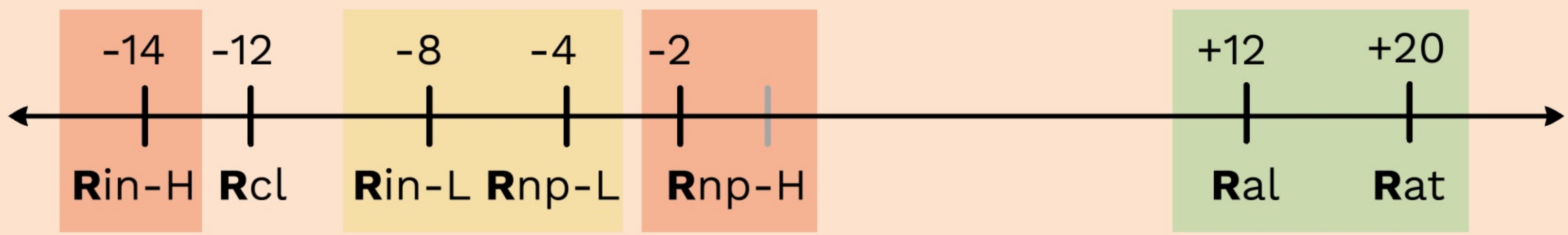
What is Reward?

Defines the **objective** of Pythia encapsulating two metrics:

- **Prefetch usefulness**
- **System feadback**

Many **different kinds** of rewards get awarded:

Accurate + **timely** (**R**at) | **Accurate** + **late** (**R**al) | **Out of physical page** (**R**cl)
No-prefetch + low/high mem b/w (**R**np-L / **R**np-H)
Inaccurate + low/high mem b/w (**R**in-L / **R**in-H)

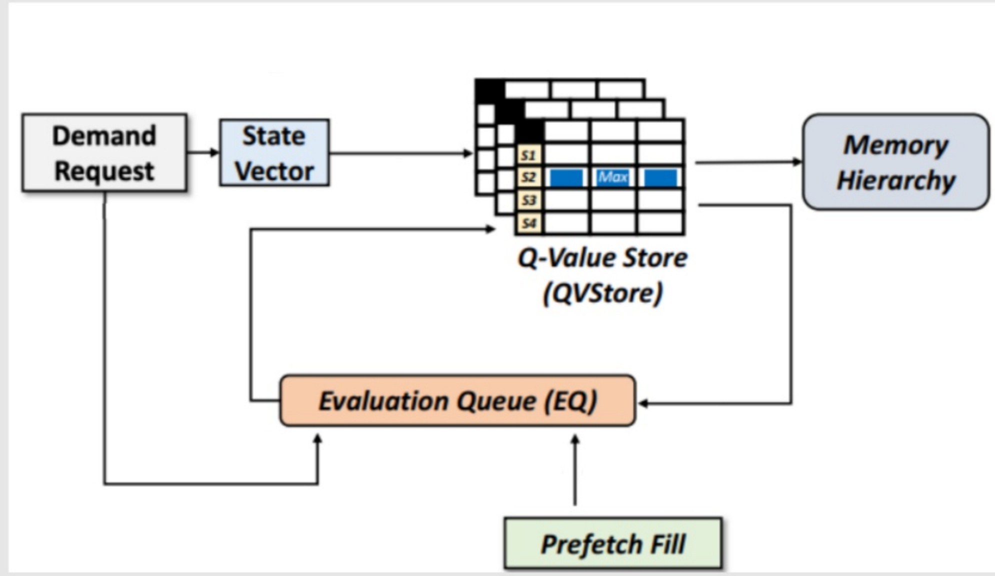


6

3. Pythia Design

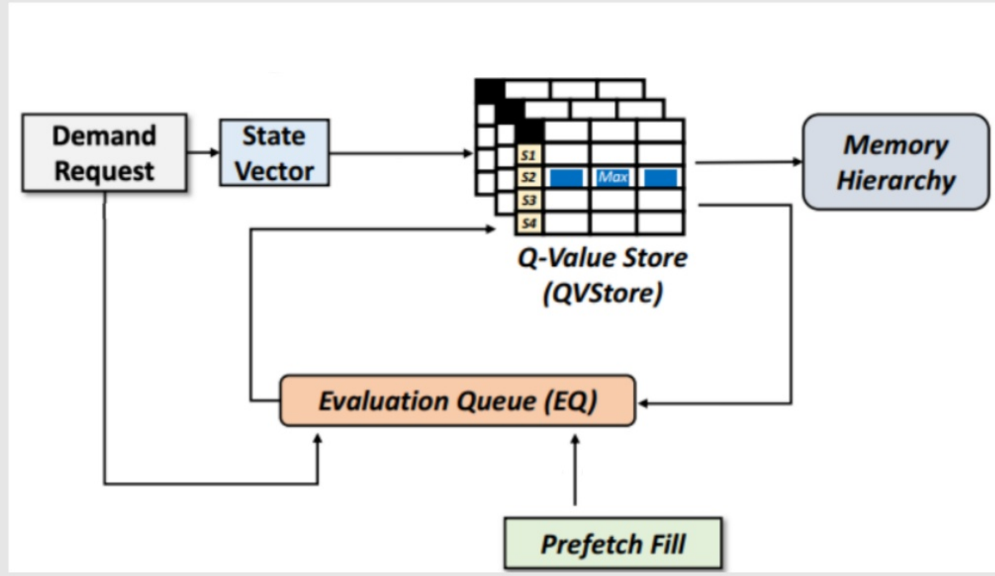
Two major components:

- **Q-Value store**
- **Evaluation Queue**



7

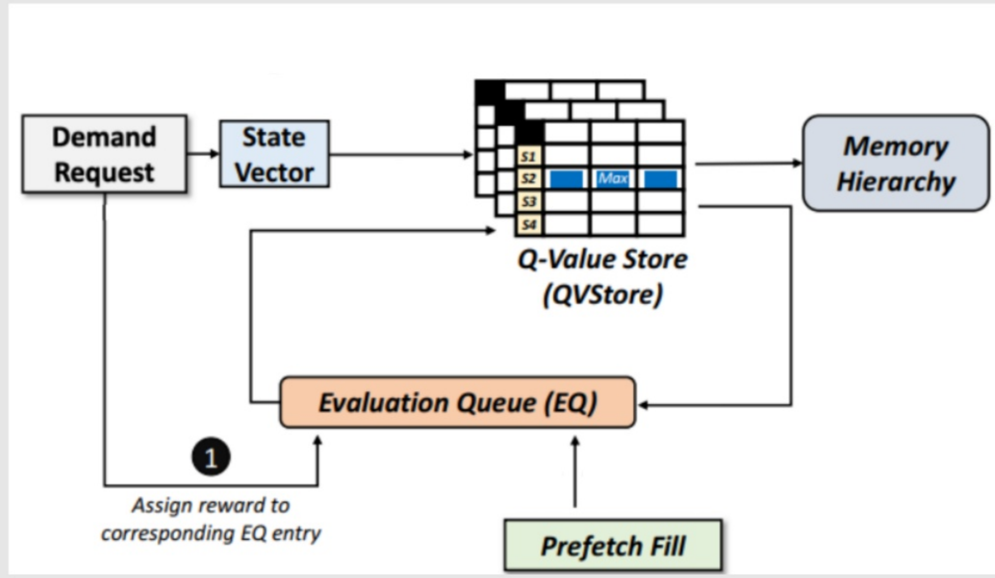
Prefetch sequence



8

Prefetch sequence

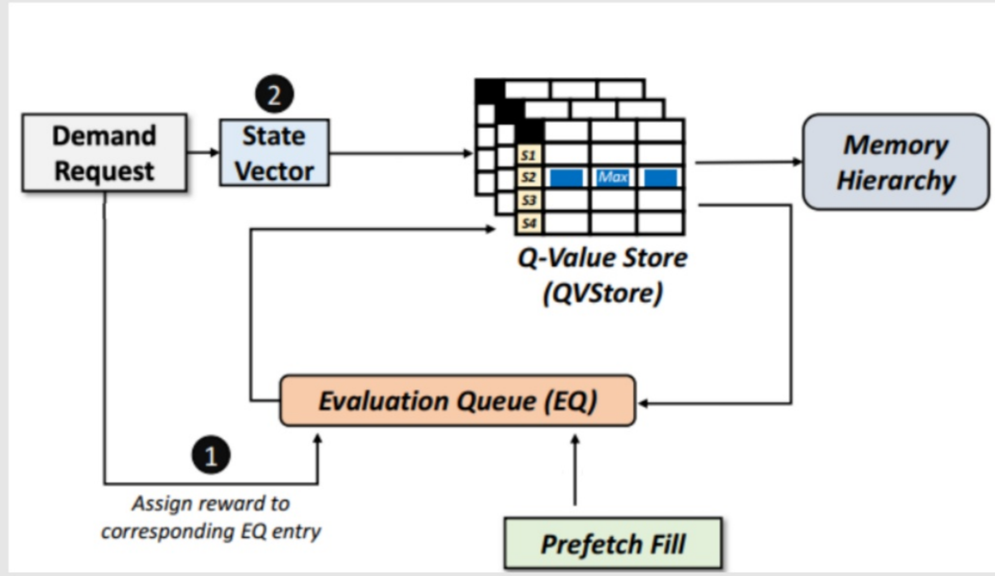
1. **Search EQ** for every new demand and assign rewards



8

Prefetch sequence

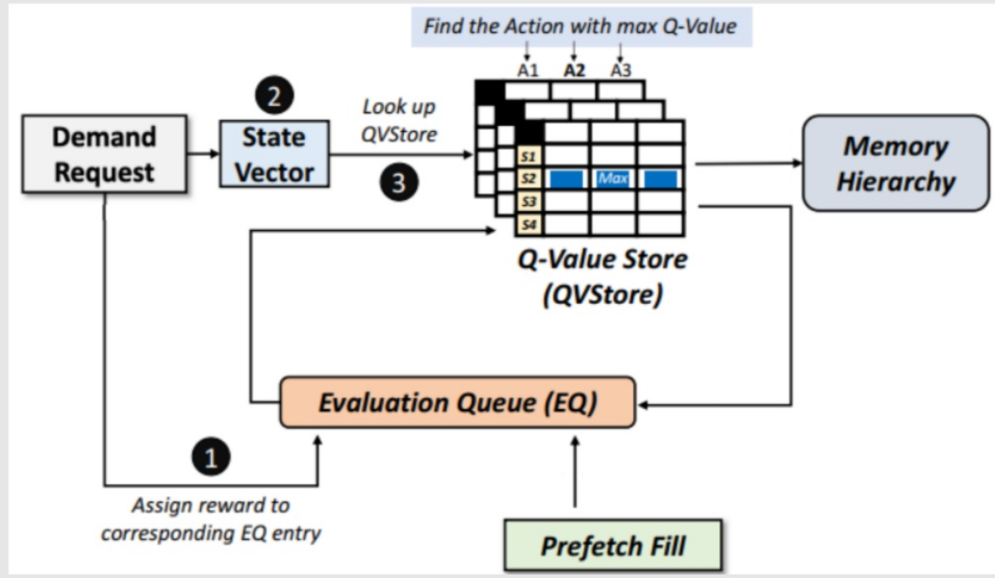
1. **Search EQ** for every new demand and assign rewards
2. **Extract** state-vector from demand



8

Prefetch sequence

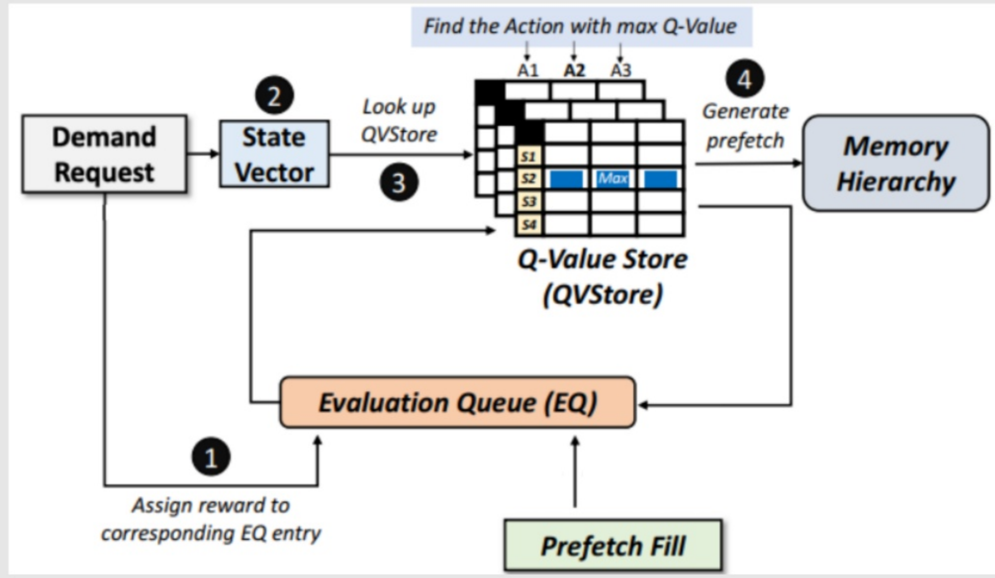
1. **Search EQ** for every new demand and assign rewards
2. **Extract** state-vector from demand
3. **Search QValue** efficiently for every possible action



8

Prefetch sequence

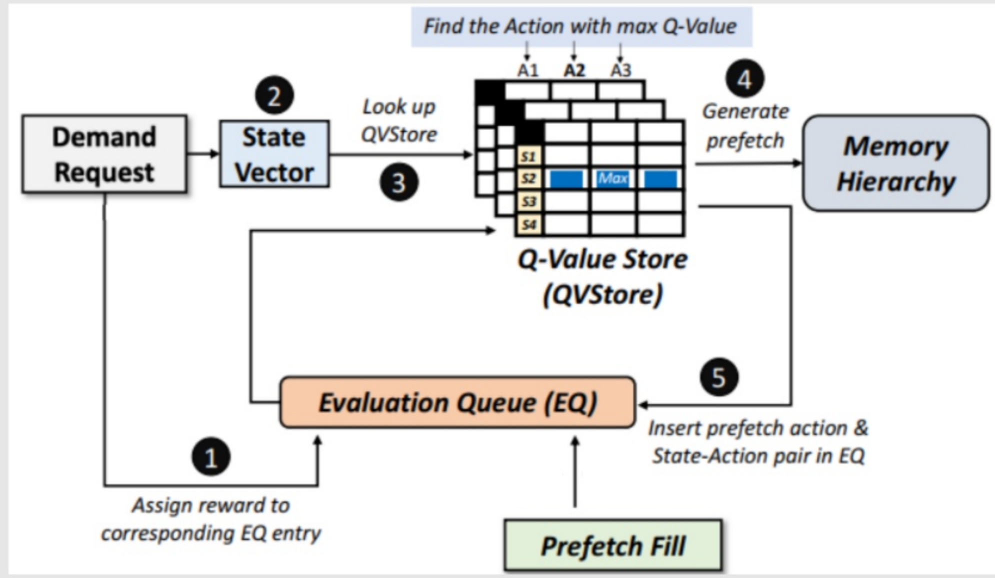
1. **Search EQ** for every new demand and assign rewards
2. **Extract** state-vector from demand
3. **Search QValue** efficiently for every possible action
4. **Issue** Memory request



8

Prefetch sequence

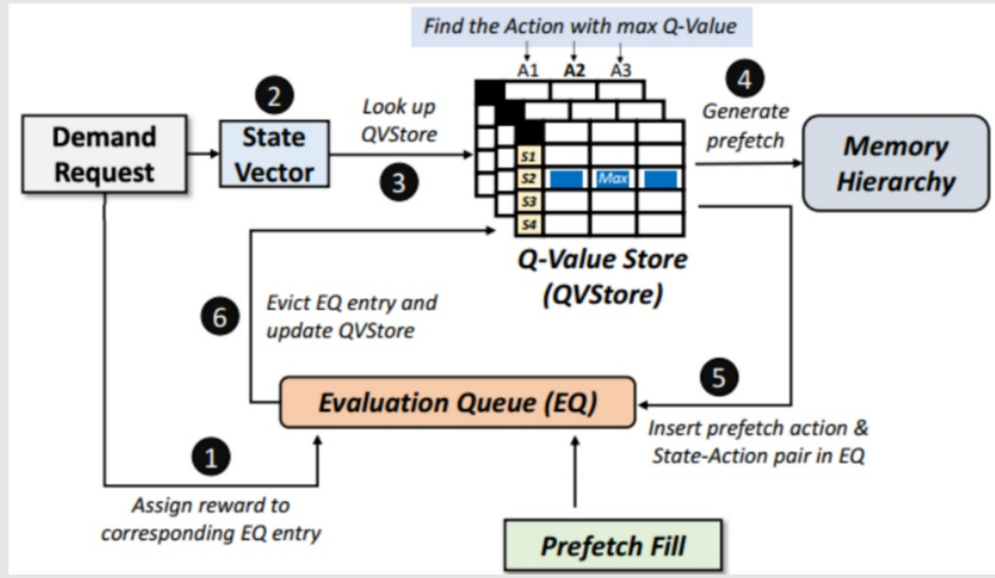
1. **Search EQ** for every new demand and assign rewards
2. **Extract** state-vector from demand
3. **Search QValue** efficiently for every possible action
4. **Issue** Memory request
5. **Add** request parameters to EQ



8

Prefetch sequence

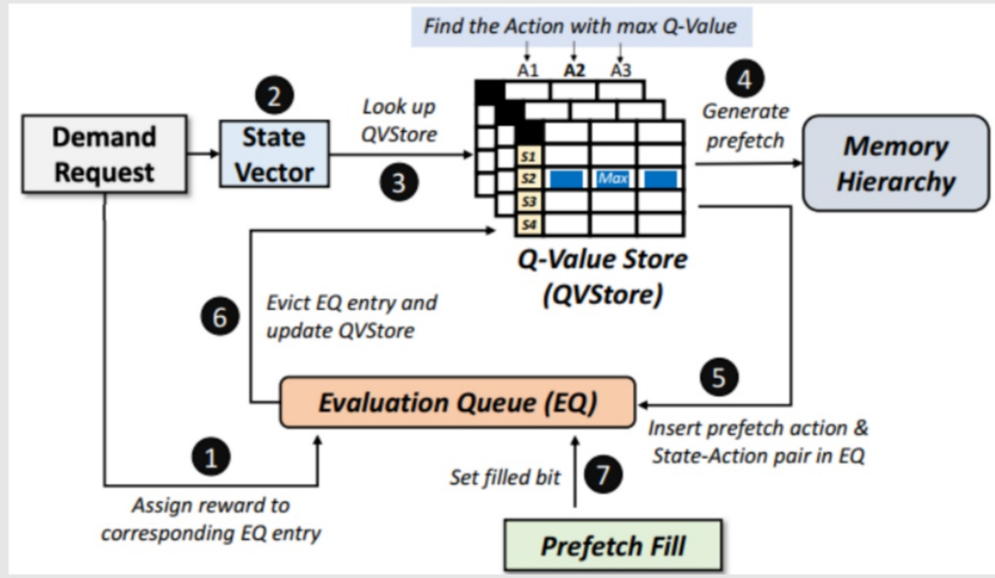
1. **Search EQ** for every new demand and assign rewards
2. **Extract** state-vector from demand
3. **Search QValue** efficiently for every possible action
4. **Issue** Memory request
5. **Add** request parameters to EQ
6. **Evict** EQ entries and update QVStore



8

Prefetch sequence

1. **Search EQ** for every new demand and assign rewards
2. **Extract** state-vector from demand
3. **Search QValue** efficiently for every possible action
4. **Issue** Memory request
5. **Add** request parameters to EQ
6. **Evict** EQ entries and update QVStore
7. When memory loads the value set **filled bit** in corresponding EQ entry



8

Organisation of QVStore

The **heart of Pythia** is the **Q-Value** store which
stores for all **state-action pairs** representing the **expected rewards**

9

Organisation of QVStore

The **heart of Pythia** is the **Q-Value** store which
stores for all **state-action pairs** representing the **expected rewards**

Instead of neural net based a **specialized 2D table** is proposed

9

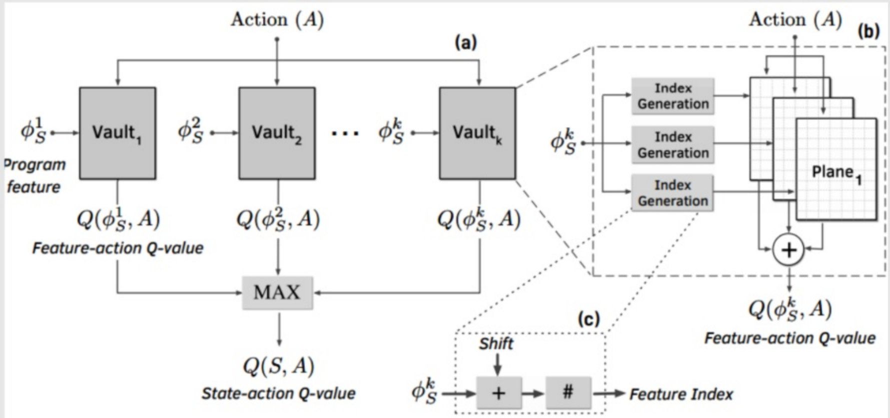
Organisation of QVStore

The **heart of Pythia** is the **Q-Value** store which
stores for all **state-action pairs** representing the **expected rewards**

Instead of neural net based a **specialized 2D table** is proposed

Problem:

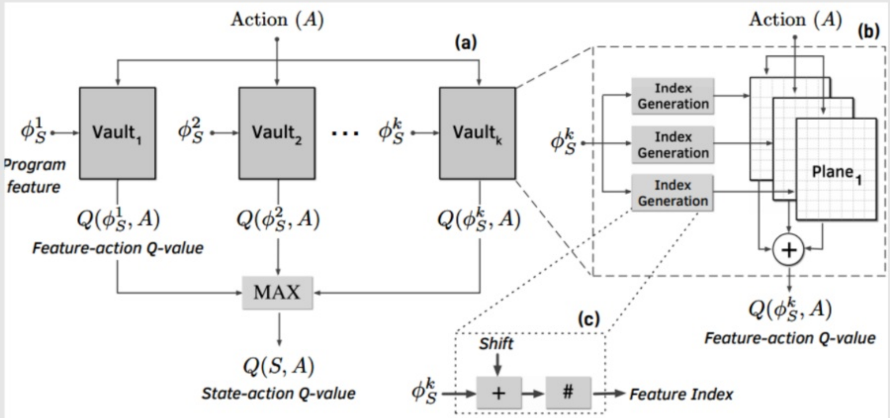
- Table size k features, hashing
- Fast search pipelining



Feature-action pairs stored in **vaults**

Multiple overlapping hash-functions implementing tile encoding

10

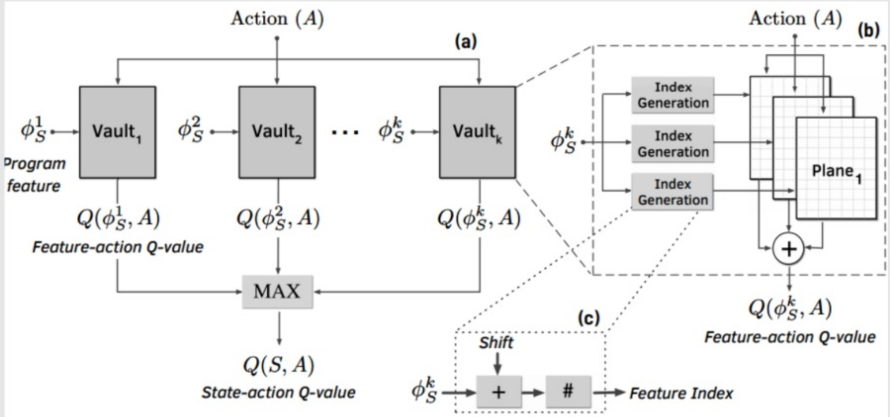


Feature-action pairs stored in **vaults**

Multiple overlapping hash-functions implementing tile encoding

=> sharing partial Q-Values for **similar features**
=> not sharing values for **wildly different features** using multiple planes

10



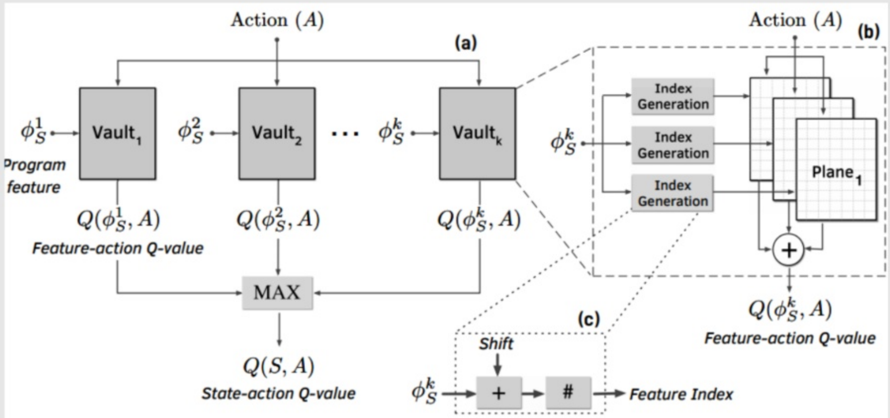
Feature-action pairs stored in **vaults**

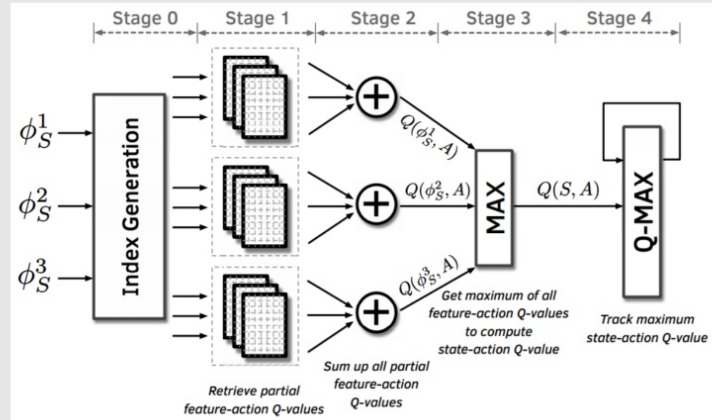
Multiple overlapping hash-functions implementing tile encoding

=> sharing partial Q-Values for **similar features**
=> not sharing values for **wildly different features** using multiple planes

Problem: need to get **max-Q** for each possible action

10





Feature-action pairs stored in **vaults**

Multiple overlapping hash-functions implementing tile encoding

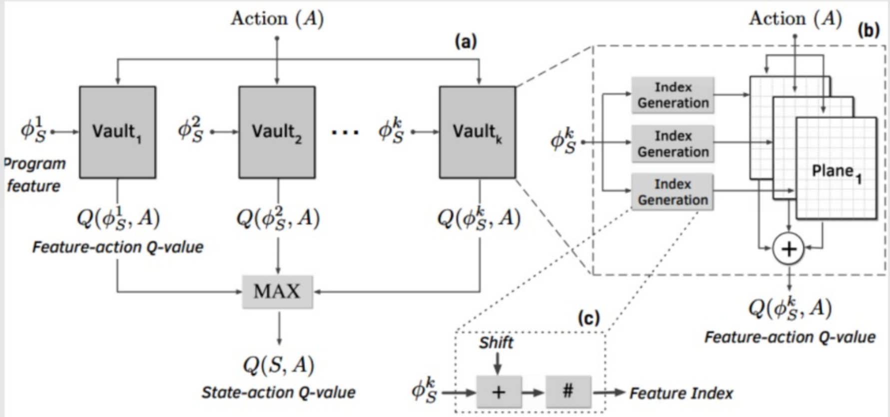
=> sharing partial Q-Values for **similar features**
=> not sharing values for **wildly different features** using multiple planes

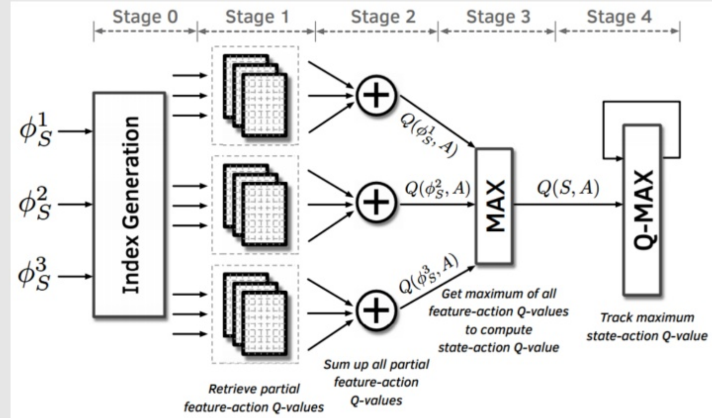
Problem: need to get **max-Q** for each possible action

Pipeline the search iterating over all possible actions

keep track of **overall max Q-Value**

10





Feature-action pairs stored in **vaults**

Multiple overlapping hash-functions implementing tile encoding

=> sharing partial Q-Values for **similar features**
=> not sharing values for **wildly different features** using multiple planes

Problem: need to get **max-Q** for each possible action

Pipeline the search iterating over all possible actions

keep track of **overall max Q-Value**

=> **drastic decrease** of critical path

=> area **overhead** stays **minimal**

10

Lots of hyperparameters

State space exploration (aka. brute forcing)

11

Lots of hyperparameters

State space exploration (aka. brute forcing)

1. Create all pairs of control-flow & data-flow components

11

Lots of hyperparameters

State space exploration (aka. brute forcing)

1. Create all pairs of control-flow & data-flow components
2. Prune list of actions [-63,63]

11

Lots of hyperparameters

State space exploration (aka. brute forcing)

1. Create all pairs of control-flow & data-flow components
2. Prune list of actions [-63,63]
3. Select best tuning configuration (uniform grid search)

11

Lots of hyperparameters

State space exploration (aka. brute forcing)

1. Create all pairs of control-flow & data-flow components
2. Prune list of actions [-63,63]
3. Select best tuning configuration (uniform grid search)

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

11

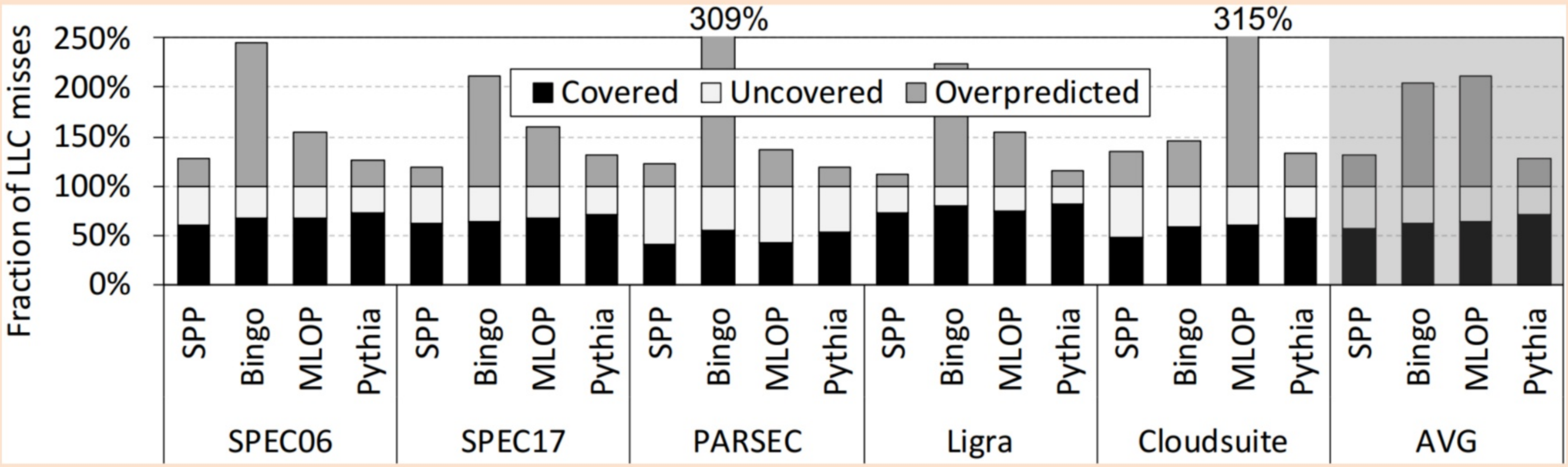
4. Performance Analysis

State of the art **prefetcher competition**:

• **SPP**	Path Confidence Lookahead	**6.2 KB**
• **Bingo**	Spatial Data Pattern	**46 KB**
• **MLOP**	Multi-Lookahead Offset	**8 KB**
• **DSPatch**	Dual Spatial Pattern	**3.6 KB**
• **PPF**	Perceptron-based Filtering	**39.3 KB**
• **Pythia**	Reinforcement Learning	**25.5 KB**

12

Coverage & Overprediction



13

Coverage & Overprediction

Pythia consistently brings the highest coverage while having lowest overpredictions

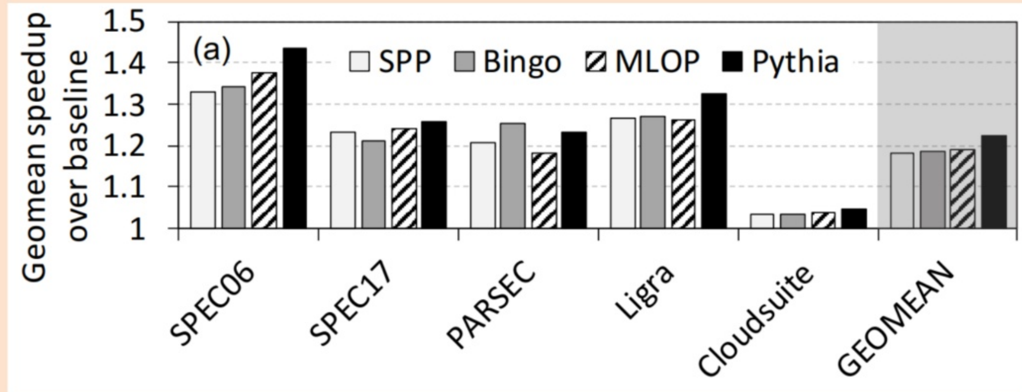
13

Workload Speedup

14

Workload Speedup

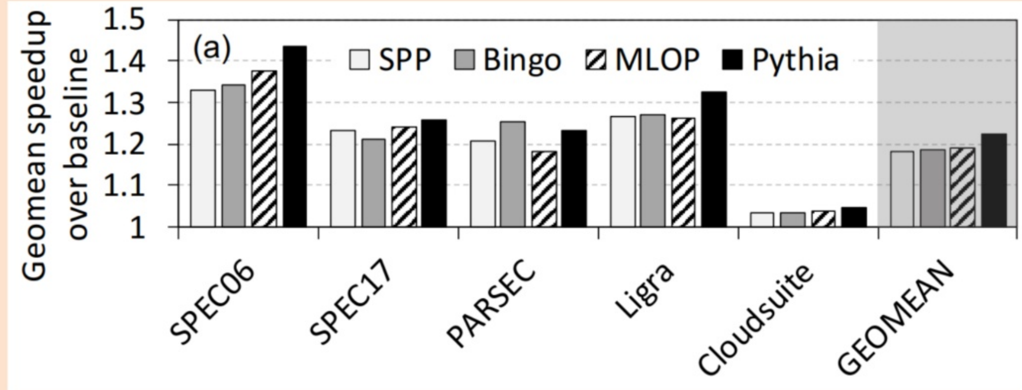
Single Core System



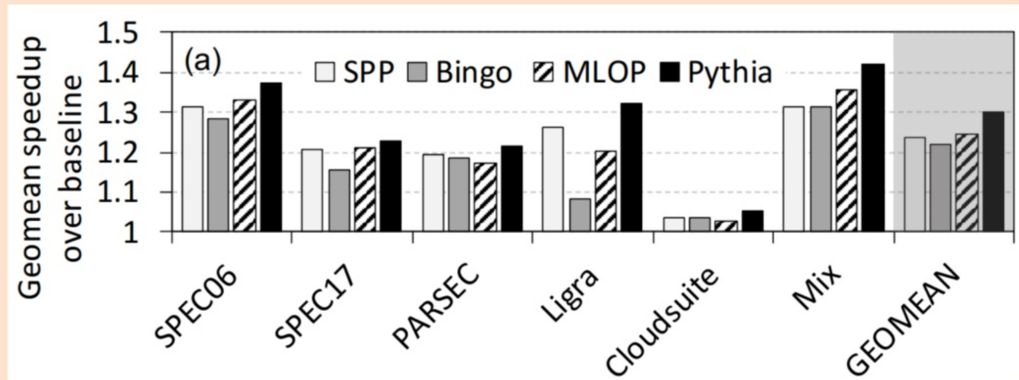
14

Workload Speedup

Single Core System



Four Core System

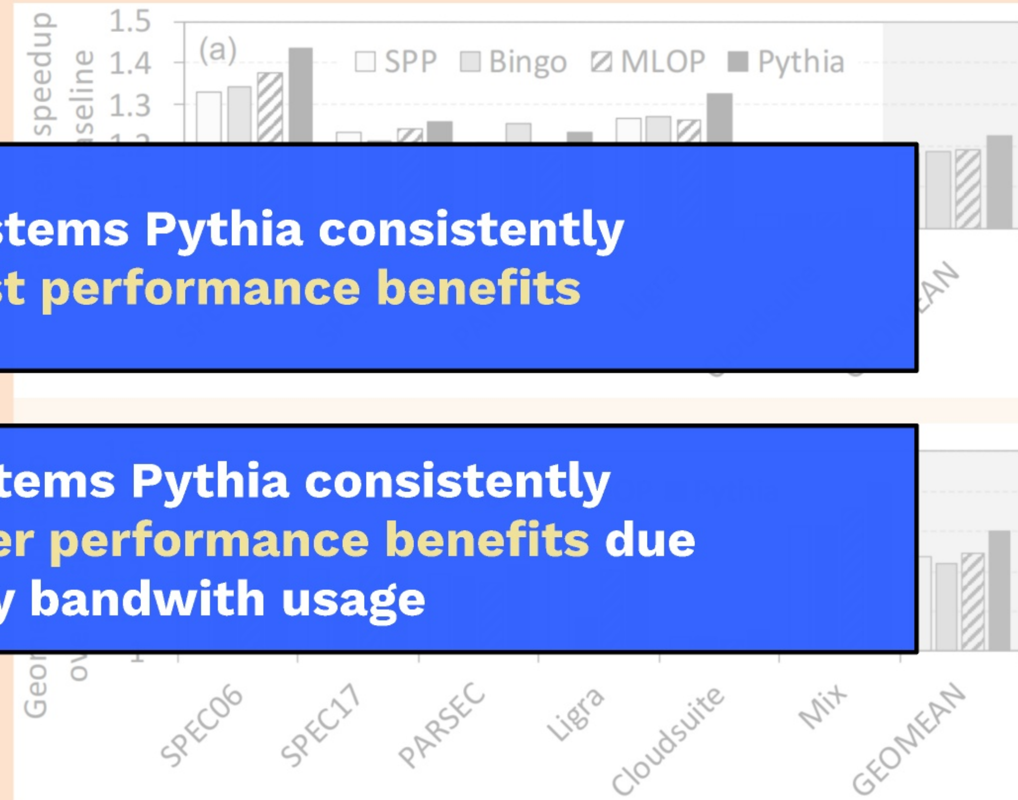


14

Workload Speedup

Single Core System

In single core systems Pythia consistently brings the highest performance benefits



In multi core systems Pythia consistently brings even higher performance benefits due to better memory bandwith usage

Four Core System

14

Executive Summary - Questions?

Background: Prefetchers predict address of future memory requests by finding access patterns from program context / feature

Problem: Three key shortcomings of prior prefetchers:

- Using only single program feature
- Lack of system awareness / feedback
- Lack of in-silicon customizability

Goal: Design adaptive and multi-feature prefetching framework

Contribution: Pythia, formulating prefetching as a reinforcement learning problem

Results:

- Evaluated using wide range of workloads
- Outperforms current best prefetchers by 3.4%, 7.7% & 17% in 1/4/bw-constrained cores

15

Strengths of the Paper

16

Strengths of the Paper

- **Simple** idea, great execution
- Multiple **levels of detail** presented
- **Intuitive** illustrations
- Good amount of self **analysis**, **reflection** conceptually and **testing**
- **Example** usage & installation

16

Weaknesses of the Paper

17

Weaknesses of the Paper

- A **lot of repetition** in the beginning
- Typical ML problem: **Only knows it works, not how!**
- **Brute forcing** its way through and no report of struggle
- Paper only states Pythia is better than everybody
  but what is the **theoretical limit** or **future improvements?**

Discussion

Are there security vulnerabilities with Prefetching as RL?

Are Prefetchers still needed with the rise of Near/In Memory Processing?

Could this Prefetcher be used in the Industry soon?
Is the simple adaption worth the benefith and overcome the "lazyness" of the industry?

Innovation instead of exploration?

18

Pythia

A Customizable
Hardware Prefetching Framework
Using Online Reinforcement Learning

Presented by
Cedric Caspar

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

ETH Zürich   Processor Architectur Research Labs, Intel Labs   TU Delft