Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Presented by: Aurelio Negri
Authors: Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, Onur Mutlu
Executive summary

- **Background:** Prefetching used for predicting future memory accesses based on past patterns and other features

- **Problem with current solutions:**
  - Using only one program feature
  - No system awareness
  - Lack of customizability
  - Memory overhead for ML models

- **Goal:**
  - Learns from multiple features and system level awareness
  - Can be customized in silicon

- **Contribution:** Online approach, customizable and practical

- **Key results:**
  - Evaluated over various workloads
  - Outperforms best prefetchers by 3.4%, 7.7% and 17%.
Outline of the presentation

• **Introduction** to prefetching
• **Limitations** with current prefetchers
• **Goals** of the current paper
• How does Pythia work
• **Results and comparisons**
• **Strengths and weaknesses**
• Discussion
What is prefetching?

• Tries to solve in part the problem of long-latency memory requests
• Based on past memory access patterns a prefetcher tries to predict future ones
• Usually, prefetcher base their predictions on a Program Feature (For example the program counter, page number…)
• Use coverage, accuracy and timeliness to evaluate and describe a prefetcher
Problems with current prefetchers.

• The paper compares Pythia to three different prefetchers: SPP, Bingo and MLOP.

• There are five fundamental problems that Pythia aims to improve:
  - Single-feature prediction
  - Lack of system awareness
  - Lack of online prefetcher design customization
  - Small model size (compared to other ML approaches)
  - Low latency predictions (compared to other ML approaches)
Goals

We want the prefetcher to be able to:

• Use *multiple program features*

• Be able to *adapt the behavior* in bandwidth intensive situations

• Can be *easily customized* on hardware for different prefetching objectives or use different features

• Small *memory overhead and latency*
Key idea: use reinforcement learning

- At each time step the environment provides the state $S_t$ to the agent.
- Agent takes an action $A_t$ based on it.
- Based on the effect of the action the agent is rewarded with $R_{t+1}$. 

![Diagram showing the interaction between Agent, Environment, State, Reward, and Action]
RL more in detail

• Agent follows a **policy** that maximizes cumulative reward

• For each action and state the agent saves a **Q-value**

• \( Q(S,A) \) is the **expected reward** in a state \( S \) for taking an action \( A \)

• Use the **SARSA** algorithm:

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

• \( \alpha \) is the **learning rate**

• \( \gamma \) is the **discount rate**

• \( \epsilon \) is the **exploration rate**
How is Pythia implemented

• State is defined as a $k$-dimensional vector of program features
• The action will be to select a prefetch offset
• There are then 5 types of rewards:
  – Accurate and timely
  – Accurate but late
  – Loss of coverage
  – Inaccurate (Low or High)
  – No-prefetch (Low or High)
How is Pythia implemented

We use two hardware structures:

• **Q-Value Store**, which records the values of all the Q(S,A)

• **Evaluation Queue** (FIFO):
  – Action taken
  – Generated prefetch address
  – Reward
  – State
  – Filled bit

• **Upon initialization** the Evaluation Queue is emptied and all the values in the QV Store are set to \((1/1-\gamma)\)
How is Pythia implemented (Algorithm)

Check for **address** was and assign **reward** accordingly.
Get the **state vector** from the system
Find the action with the max Q-value inside the QV Store
How is Pythia implemented (Algorithm)

No-prefetch or out of page we can be inserted
If we have a prefetch action, we can generate the prefetch address.
How is Pythia implemented (Algorithm)

Update the EQ

1. Assign reward to corresponding EQ entry
2. State Vector
3. Look up QVStore
4. Generate prefetch
5. Find the Action with max Q-Value
6. Insert prefetch action, State-Action pair in EQ, Generated prefetch address
7. Evict EQ entry and update QVStore
8. Set filled bit
Evict the last entry of EQ and update the QV Store, also assign rewards for misses

Reminder:
\[ Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \]
How is Pythia implemented (Algorithm)

Set bit when cache gets filled
Pseudocode:

Algorithm 1 Pythia’s reinforcement learning based prefetching algorithm

1: procedure Initialize 
2: initialize QVStore: \( Q(S,A) \leftarrow \frac{1}{T} \)
3: clear EQ
4:
5: procedure Train_and_Predict(Addr) /* Called for every demand request */
6: entry \leftarrow search_EQ(Addr) /* For a demand request to Addr, search EQ with the demand address */
7: if entry is valid then
8: if entry.filled == true then
9: entry.reward \leftarrow R_{AT} /* If the filled bit is set, i.e., the demand access came after the prefetch fill, assign reward R_{AT} */
10: else
11: entry.reward \leftarrow R_{AL} /* Otherwise, assign R_{AL} */
12: S \leftarrow get_state() /* Extract the state-vector from the attributes of current demand request */
13: if rand() \leq \epsilon then
14: action \leftarrow get_random_action() /* Select a random action with a low probability \( \epsilon \) to explore the state-action space */
15: else
16: action \leftarrow argmax_a Q(S,a) /* Otherwise, select the action with the highest Q-value */
17: prefetch(Addr + Offset[action]) /* Add the selected prefetch offset to the current demand address to generate prefetch address */
18: entry \leftarrow create_EQ_entry(S, action, Addr + Offset[action]) /* Create new EQ entry using the current state-vector, the selected action, and the prefetch address */
19: if no prefetch action then
20: entry.reward \leftarrow R_{NP}^H or R_{NP}^L /* In case of no-prefetch action, immediately assign reward R_{NP}^H or R_{NP}^L based on current memory bandwidth usage */
21: else if out-of-page prefetch then
22: entry.reward \leftarrow R_{CL} /* In case of out-of-page prefetch action, immediately assign reward R_{CL} */
23: evict_entry \leftarrow insert_EQ(entry) /* Insert the entry. Get the evicted EQ entry. */
24: if has_reward(dq_entry) == false then
25: dq_entry.reward \leftarrow R_{IN}^H or R_{IN}^L /* If the evicted entry does not have a reward yet, assign the reward R_{IN}^H or R_{IN}^L based on current memory bandwidth usage */
26: R \leftarrow dq_entry.reward /* Get the reward stored in the evicted entry */
27: S_{1} \leftarrow dq_entry.state; A_{1} \leftarrow dq_entry.action /* Get the state-vector and the action from the evicted EQ entry */
28: S_{2} \leftarrow EQ.head.state; A_{2} \leftarrow EQ.head.action /* Get the state-vector and the action from the entry at the head of the EQ */
29: \[ Q(S_{1},A_{1}) \leftarrow Q(S_{1},A_{1}) + \alpha[R + yQ(S_{2},A_{2}) - Q(S_{1},A_{1})] \] /* Perform the SARSA update */
30:
31: procedure Prefetch_Fill(Addr) /* For every prefetch fill, search the address in EQ and mark the corresponding EQ entry as filled */
32: search_and_mark_EQ(Addr, FILLED)
How does the QV Store works

• Each program feature we use has a vault
• Each vault consist in several (current implementation 3) planes
• A plane is essentially a bidimensional array that spans action × feature space.
• Perform tile encoding by using a hash function on the feature value
Q-value extraction in detail
Q-Max extraction value
Design-Space exploration

We need to define all the **program features** we want to use, the action set, **reward levels** and **hyperparameters** to run Pythia.

<table>
<thead>
<tr>
<th>Features</th>
<th>PC+Delta, Sequence of last-4 deltas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefetch Action List</td>
<td>{-6,-3,-1,0,1,3,4,5,10,11,12,16,22,23,30,32}</td>
</tr>
<tr>
<td>Reward Level Values</td>
<td>( R_{AT}=20, \ R_{AL}=12, \ R_{CL}=-12, \ R_{IN}^H=-14, \ R_{IN}^L=-8, \ R_{NP}^H=-2, \ R_{NP}^L=-4 )</td>
</tr>
<tr>
<td>Hyperparameters</td>
<td>( \alpha = 0.0065, \gamma = 0.556, \epsilon = 0.002 )</td>
</tr>
</tbody>
</table>
Area and power overhead

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

Pythia’s area: 0.33 mm$^2$/core; Pythia’s power: 55.11 mW/core
# Storage overhead

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
</table>
| **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** |
Comparison of storage overhead

<table>
<thead>
<tr>
<th>Method</th>
<th>Features</th>
<th>Storage Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPP [78]</td>
<td>256-entry ST, 512-entry 4-way PT, 8-entry GHR</td>
<td>6.2 KB</td>
</tr>
<tr>
<td>Bingo [27]</td>
<td>2KB region, 64/128/4K-entry FT/AT/PHT</td>
<td>46 KB</td>
</tr>
<tr>
<td>MLOP [111]</td>
<td>128-entry AMT, 500-update, 16-degree</td>
<td>8 KB</td>
</tr>
<tr>
<td>DSPatch [30]</td>
<td>Same configuration as in [30]</td>
<td>3.6 KB</td>
</tr>
<tr>
<td>PPF [32]</td>
<td>Same configuration as in [32]</td>
<td>39.3 KB</td>
</tr>
<tr>
<td>Pythia</td>
<td>2 features, 2 vaults, 3 planes, 16 actions</td>
<td>25.5 KB</td>
</tr>
</tbody>
</table>
Results and comparisons

Speedup in single and four-core systems across various workloads

Performs better for almost all workloads

Customization can give another 2% to 7.8% improvement
Results and comparisons

Speedups over multiple cores:

Pythia outperforms the other prefetchers in all core configurations.

The gain increases with proportionally to the number of cores.
Results and comparisons

Speedups between different # transfer per seconds in the DRAM:

Pythia outperforms also in a wide range of bandwidth configurations.
Strengths

- Ability to take into consideration the memory bandwidth
- Dynamically adapt the behavior depending on the workload
- Well optimized and scalable implementation of a machine learning algorithm
- Possibility and ease of customization
- Applicability of the concept in other computer architecture areas
Weaknesses

- No latency comparison with other prefetchers
- No tradeoff analysis between a larger model implementation and a performance gain
- Reinforcement learning needs training before it can be use effectively
Conclusions

- **Background**: Prefetching used for predicting future memory accesses based on past patterns and other features

- **Problem with current solutions**:  
  - Using only one program feature  
  - No system awareness  
  - Lack of customizability  
  - Memory overhead for ML models

- **Goal**:  
  - Learns from multiple features and system level awareness  
  - Can be customized in silicon

- **Contribution**: Online approach, customizable and practical

- **Key results**:  
  - Evaluated over various workloads
  - Outperforms best prefetchers by 3.4%, 7.7% and 17%.
Discussion (Training time)

- How can we improve it on a **single core** system?
  - Try different initialization values
- And on a **multi core**?
  - Share the content of the QV Store
- How do we manage **context switching**?
  - Split the EQ and QV Store for each thread?
  - Is it actually a bad thing that the information is shared?
Discussion (Applicability)

• Where else can we apply the same concept of RL?
  ➢ Branch prediction
  ➢ Switch between different energy modes (i.e., Low Power Mode vs. Performance Mode)
  ➢ Adaptive bitrate streaming
Discussion (Other ML methods)

• What are other Machine Learning methods that we can use?
  ➢ Supervised learning (Neural networks)

• How can we transform Pythia into a Neural Network?
  ➢ Standard hidden layer + SoftMax layer with the actions
  ➢ Need to find a way to create the right input data

• Other discussion points?