pa2-lehra-do-prefetcher-team-gandiva

Task 1: TLB Prefetching

Implementing the Arbitrary Stride Prefetcher

The objective of this implementation was to develop an **Arbitrary Stride Prefetcher (ASP)** for the **Shared Translation Lookaside Buffer (STLB)**. Below is a detailed walkthrough of the implementation process:

1. **Defining the IP Tracker Class**

The ASP makes use of a class named `IP_TRACKER` to maintain records for each unique instruction pointer (IP). The class is defined with the following attributes:

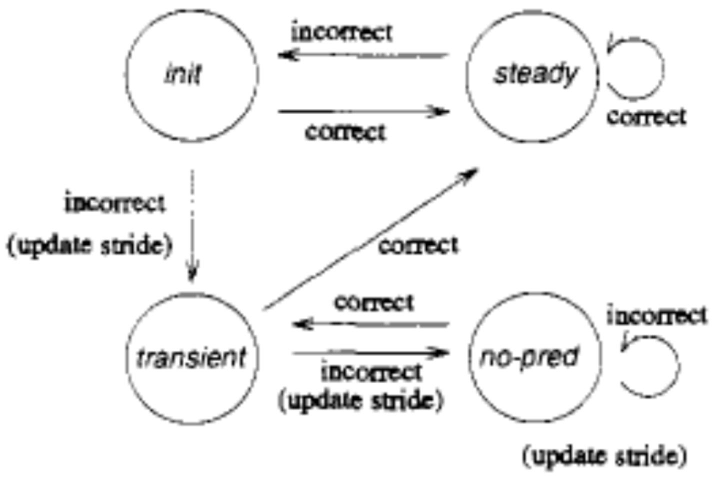
- `ip`: Stores the instruction pointer value.
- `last_addr`: Holds the most recent address accessed by this IP.
- `last_stride`: Stores the stride calculated as the difference between consecutive addresses.
- `state`: Represents the state of the tracker in the stride prediction state machine.
- `access_time`: Keeps track of the last access time, used for the **Least Recently Used (LRU)** replacement policy.

This class is instantiated for each of the IPs tracked by the prefetcher, and a total of 64 such trackers are maintained.

2. **Prefetching Logic in the `stlb_prefetcher_operate()` Function**

The core logic is handled within the `stlb_prefetcher_operate()` function, which performs the following steps for each address accessed:

1. **Identify or Allocate IP Tracker**:
    - Search for the IP in the existing tracker table.
    - If the IP is not found and all trackers are in use, apply an **LRU replacement** to replace the least recently accessed tracker.
    - Initialize the tracker fields: `ip`, `last_addr`, `last_stride`, and set `state` to `INITIAL`.
2. **Calculate the Stride**:
    - Compute the stride as the difference between the current and previous address.
3. **State Machine Logic**:
    - The stride consistency is monitored through a state machine with four states:
        - **INITIAL**: The first reference; initializes the stride pattern.
        - **TRANSIENT**: If the stride pattern changes, temporarily mark as inconsistent.
        - **STEADY**: If the same stride is observed consecutively, mark the pattern as stable.
        - **NOPRED**: No reliable prediction; monitor until consistent strides are observed.
    - **Figure 1: State machine implemented:**



1. **Prefetching**:
   - Prefetching is initiated only if the stride pattern is in the `STEADY` state.
   - The number of prefetches is determined by a parameter called `PREFETCH_DEGREE`.
   - For each calculated prefetch address, the `prefetch_translation()` function is called to prefetch the addresses into the STLB.

3. **Final Statistics Collection**

The `stlb_prefetcher_final_stats()` function outputs a summary of the prefetcher's performance. It reports the final prefetch degree and any other relevant statistics.

Building and Running the Prefetcher

Build Command

To build the prefetcher with the required configuration:

```
# Navigate to the ChampSim directory
cd path/to/champsim

./build_champsim.sh no asp 1
```

This command specifies:

- `no`: No additional optimizations or configurations.
- `asp`: The Arbitrary Stride Prefetcher for the STLB.
- `1`: Number of CPU cores to use.

Run Command

To execute the binary with the appropriate parameters:

```
./bin/no-asp-1core -warmup_instructions 25000000 -simulation_instructions
25000000 -traces given/traces/trace1.champsimtrace.xz > output/task1/no-
asp-1core-degree-8.log
```

This command runs the simulator with:

- **Warmup Instructions**: 25,000,000
- **Simulation Instructions**: 25,000,000
- **Trace File**: `trace1.champsimtrace.xz`
- **Output Log**: Stores the results in `output/task1/no-asp-1core-degree-8.log`

Experimental Results

1. Speedup Analysis

The speedup is calculated as:

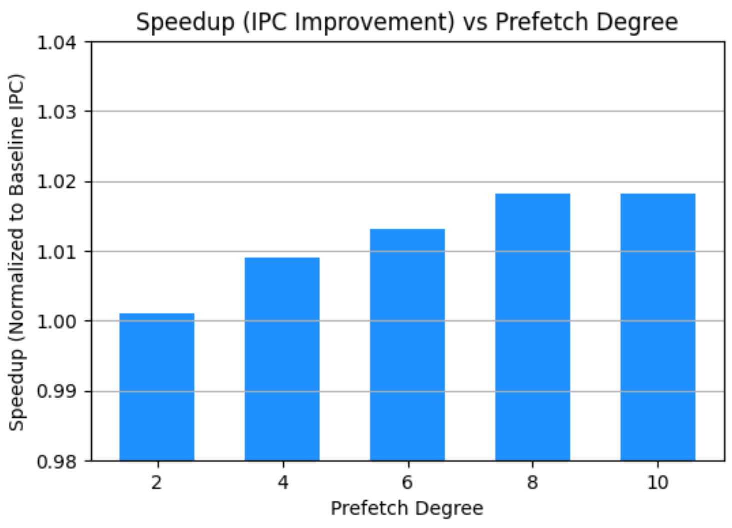
Speedup = (IPC of Prefetcher) / (IPC of Baseline without Prefetching)

We varied the **Prefetch Degree** from 2 to 10 and observed the effect on IPC. The following graph depicts the **Speedup vs. Prefetch Degree**.

Table 1: STLB Speedup Comparison

Prefetch Degree	ASP Prefetcher Speedup
2	1.001
4	1.009
6	1.013
8	1.018
10	1.018

Figure 2: Speedup vs. Prefetch Degree



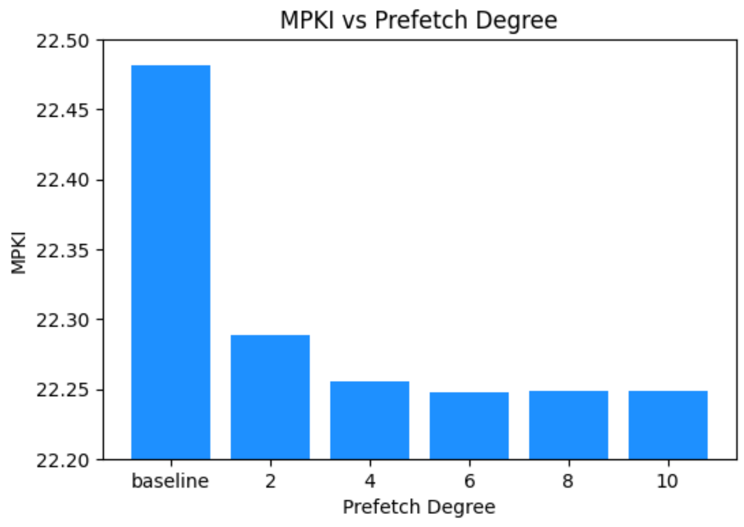
2. STLB MPKI Analysis

STLB MPKI (Misses Per Kilo Instructions) was calculated for different Prefetch Degrees and compared with the baseline where no prefetching was used. The results are summarized in the table below:

Table 2: STLB MPKI Comparison

Prefetch Degree	Baseline (No Prefetching)	ASP Prefetcher MPKI
2	22.4811	22.2884
4	22.4811	22.2556
6	22.4811	22.2476
8	22.4811	22.2491
10	22.4811	22.2491

Figure 3: STLB MPKI Comparison Graph



3. **Key Observations**

- The **Speedup** peaked at **Prefetch Degree = 8** and statyed constant beyond that point.
- The **STLB MPKI** decreased to minimum at Degree 6, indicating fewer misses and improved prefetching efficiency. But after this point it remains mostly constatnt as extra prefecth requests were dropped by the pre fecther itself.

Conclusion

The Arbitrary Stride Prefetcher (ASP) successfully reduced STLB misses and improved the overall performance. Optimal performance was observed at **Prefetch Degree 8**, where both speedup and MPKI were optimized. Further tuning of the state machine and replacement policies may provide additional improvements.

Task 2: Data Prefetcher

IP-Stride and Complex-Stride Prefetcher Implementation Analysis

2.1 Overview of Prefetcher Designs

Two types of stride-based prefetchers were implemented and evaluated in the **ChampSim** simulation environment: **IP-Stride Prefetcher** and **Complex-Stride Prefetcher**. This section outlines the step-by-step procedure for implementing each prefetcher, along with performance analysis and comparisons.

2.2 Implementation Steps

2.2.1 IP-Stride Prefetcher Implementation

1. **Implement the Prefetcher Logic**:
   - The IP-Stride prefetcher tracks stride patterns for each instruction pointer (IP) and issues prefetches based on consistent strides.
   - State machine for tracking `INITIAL`, `TRANSIENT`, `STEADY`, and `NOPRED` states.
   - Prefetches are issued within the same 4KB page to avoid cross-page pollution.

2. **Build and Execute**:
   - Use the following commands to build and execute the prefetcher:

```
# Navigate to the ChampSim directory
cd path/to/champsim

# Build the IP-Stride Prefetcher
./build_champsim.sh ip_stride no 1

# Run the simulation with the given trace and configurations
 ./bin/ip_stride-no-1core -warmup_instructions 25000000 -
simulation_instructions 25000000 -traces
../given/traces/trace2.champsimtrace.xz
```

> **Key Takeaways**

IP-Stride prefetcher is a basic yet efficient mechanism for leveraging consistent stride patterns, making it suitable for workloads dominated by sequential memory access. However, it may fall short in scenarios with non-linear or complex access patterns, where more advanced prefetching strategies like the Complex-Stride prefetcher would be more effective.

2.2.2 Complex-Stride Prefetcher Implementation

1. **Implement the Prefetcher Logic**:
   - The Complex-Stride Prefetcher extends the IP-Stride by incorporating **delta-strides**.
   - Track stride changes **pattern** for each IP, using a **confidence metric** to gauge prediction reliability.

  - Implement a signature-based indexing mechanism for storing and retrieving complex stride patterns.

2. **Build and Execute**:

  - Use the following commands to build and execute the Complex-Stride Prefetcher:

```
# Build the Complex-Stride Prefetcher
./build_champsim.sh complex_stride no 1

# Run the simulation with the given trace and configurations
./bin/complex_stride-no-1core -warmup_instructions 25000000 -
simulation_instructions 25000000 -traces
../given/traces/trace2.champsimtrace.xz
```

> **Key Takeaways**

This implementation showcases a robust stride prefetcher that can handle both regular and complex stride patterns, making it well-suited for modern applications with diverse memory access behaviors.

2.3 Performance Metrics and Evaluation

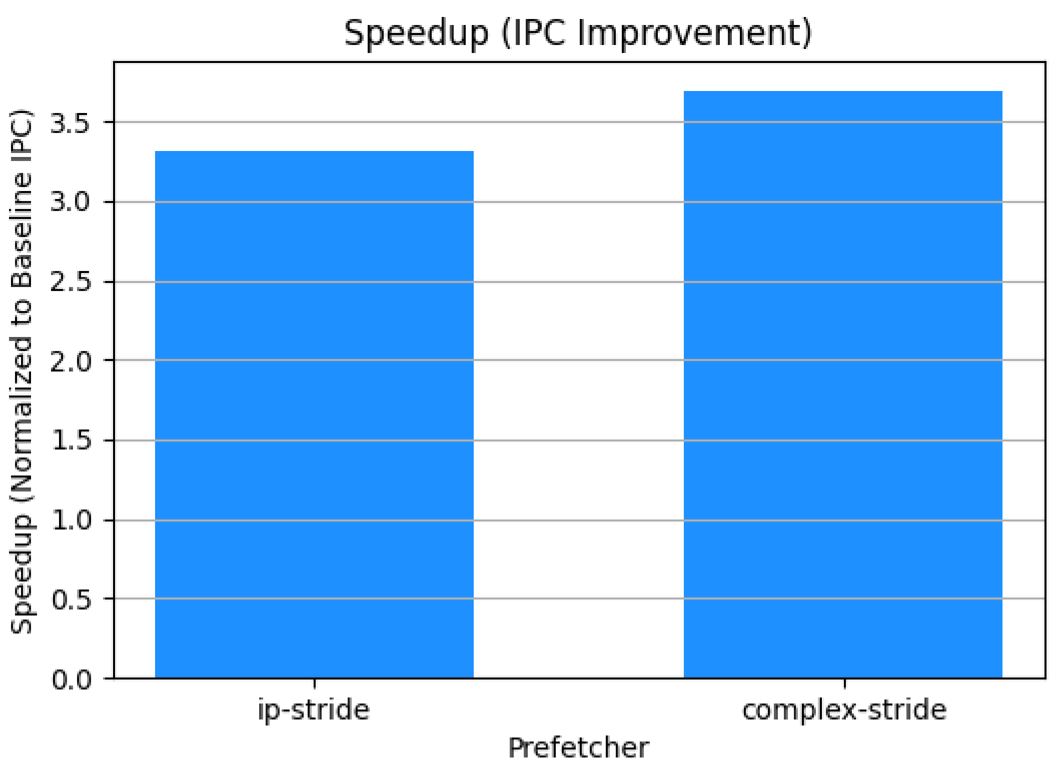
The performance of the implemented prefetchers was evaluated against a **baseline (no prefetcher)**. The key metrics used for comparison are:

2.3.1 Speedup Analysis

- **IP-Stride Speedup**: IP-Stride was compared to the baseline and achieved a speedup of `230.9%` for the given trace.
- **Complex-Stride Speedup**: Complex-Stride prefetcher outperformed the baseline and IP-Stride, with a speedup of `269.5%`.

Prefetcher	Speedup (normalized)
No Prefetcher	1
IP-Stride	3.309
Complex-Stride	3.695

- **Plot Placeholder**:

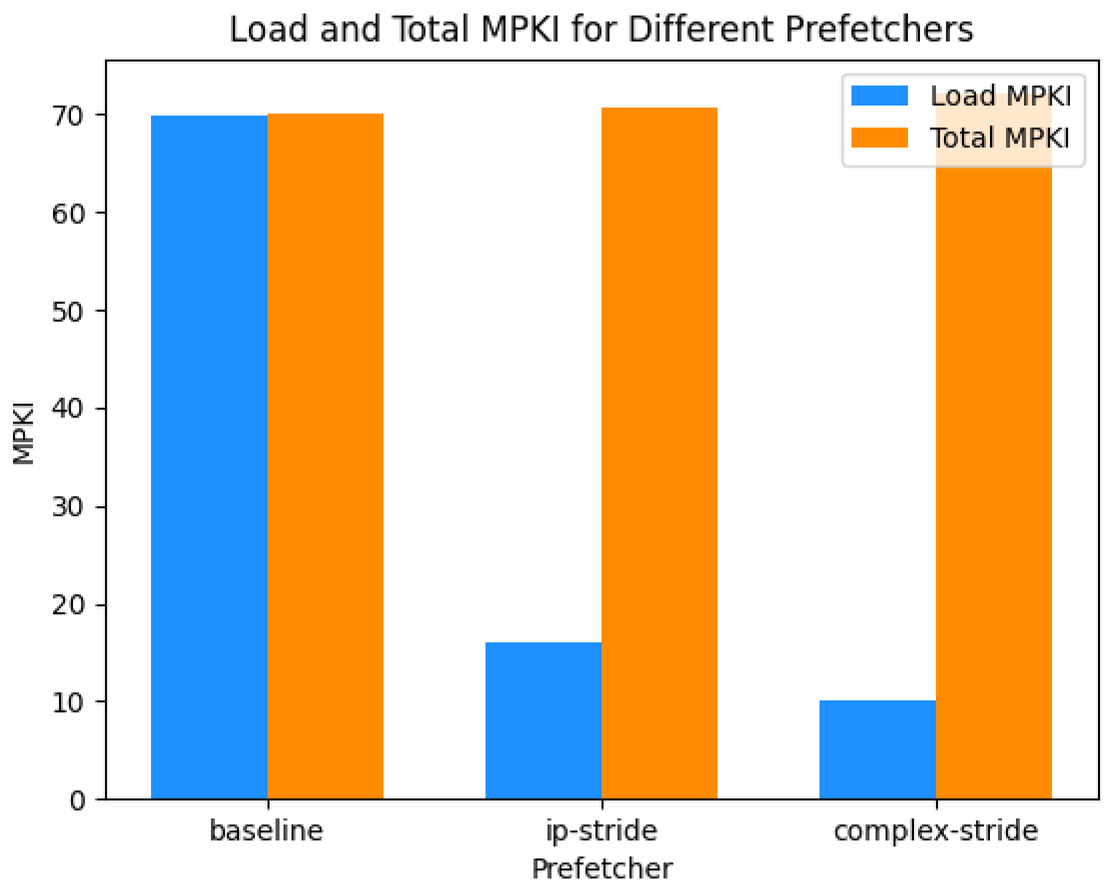


2.3.2 L1D MPKI Analysis

- **Baseline**: Without any prefetching, the L1D Load Misses Per 1000 Instructions (MPKI) was observed to be `69.9647`.
- **IP-Stride**: The IP-Stride prefetcher reduced L1D MPKI by `77%` over the baseline.
- **Complex-Stride**: The Complex-Stride prefetcher demonstrated a more significant reduction of `85.7%` in L1D MPKI compared to IP-Stride and baseline.

Prefetcher	L1D MPKI (Load)	L1D MPKI (Total)
No Prefetcher	69.9647	69.9934
IP-Stride	16.067	70.7358
Complex-Stride	9.98436	72.0802

- **Plot**:



2.4 Comparative Analysis

The results show that **Complex-Stride Prefetcher** performs better than the **IP-Stride Prefetcher** for the given trace2, with lower L1D MPKI and higher speedup. The additional complexity of handling strides' **historical patterns** and maintaining a confidence-based state tracking mechanism allows the Complex-Stride Prefetcher to better adapt to varying memory access patterns.

2.5 Conclusion

The **Complex-Stride Prefetcher** outperforms the **IP-Stride Prefetcher** and the **Baseline** in terms of both **speedup** and **L1D MPKI reduction** in case of running trace2. This highlights its effectiveness in handling complex memory access patterns, making it suitable for workloads with irregular and non-linear memory references.

Task 3: Guldasta-e-Prefetcher

Objective

In this task, the goal was to implement a **hybrid prefetching strategy** that dynamically selects the best prefetcher for a given workload from a set of candidate prefetchers. The candidate prefetchers are:

1. **IP-Stride Prefetcher**: Tracks fixed stride patterns for each instruction pointer.
2. **Complex-Stride Prefetcher**: Extends IP-Stride by considering delta-strides and a confidence-based state machine.
3. **Next-Line Prefetcher**: Prefetches the next consecutive cache lines.

Implementation Overview

The hybrid strategy, called **Guldasta-e-Prefetcher**, works by evaluating the accuracy of each prefetcher during a **learning phase** and selecting the most accurate one for the rest of the simulation. This ensures optimal performance across diverse memory access patterns. The following steps outline the implementation process:

1. Prefetcher Selection Strategy

The strategy involves:

1. **Learning Phase**:
    - During the initial phase after warmup, run all three prefetchers (`IP-Stride`, `Complex-Stride`, `Next-Line`) for a fixed **`PHASE_LENGTH`** (number of prefetch requests).
    - Track the accuracy of each prefetcher using the ratio of useful prefetches to total prefetches.
2. **Accuracy Measurement**:
    - Define accuracy as: Accuracy = Number of Prefetch Hits / Total Number of Prefetches
3. **Prefetcher Selection**:
    - At the end of the learning phase, choose the prefetcher with the highest accuracy.
    - Switch to the selected prefetcher for the remaining simulation.

2. Code Modifications

2.1 `optimized.l1d_pref` Implementation

1. **PHASE_LENGTH Parameter**:
    - Took a `PHASE_LENGTH` value of `10000` load instructions to balance between learning and execution phases.
2. **Prefetcher Classes**:
    - Implement separate counters for each prefetcher (`IPStride`, `ComplexStride`, `NextLine`) and a counter tracking demand misses
    - Track each prefetcher's hits during the learning phase and calculate coverage after phase ends.
3. **Prefetcher Selector**:
    - Adds a method `get_current_prefetcher` to `CACHE` class that handles the learning phase, switching prefetchers, and choose the best prefetcher based on observed coverage.

3. Build and Run Instructions

To build and execute the hybrid prefetcher, follow these steps:

1. **Build Command:**

```
# Navigate to the ChampSim directory
cd path/to/champsim

# Build the optimized prefetcher with Guldasta-e-Prefetcher
./build_champsim.sh optimized no 1
```

2. **Run Command:**

```
# Run the binary with trace1, trace2, and trace3 for evaluation

./bin/optimized-no-1core -warmup_instructions 25000000 -
simulation_instructions 25000000 -traces
../given/traces/trace1.champsimtrace.xz

./bin/optimized-no-1core -warmup_instructions 25000000 -
simulation_instructions 25000000 -traces
../given/traces/trace2.champsimtrace.xz

./bin/optimized-no-1core -warmup_instructions 25000000 -
simulation_instructions 25000000 -traces
../given/traces/trace3.champsimtrace.xz
```

4. Experimental Results

The performance of each prefetcher was evaluated across three traces (`trace1`, `trace2`, and `trace3`). The key metrics considered are **Speedup** and **L1D MPKI**.

4.1 Speedup Analysis

Speedup is defined as:

Speedup = IPC of Prefetcher / PC of Baseline without Prefetching

We compare the performance of the four prefetchers:

1. **Baseline (No Prefetcher)**
2. **IP-Stride Prefetcher**
3. **Complex-Stride Prefetcher**
4. **Next-Line Prefetcher**
5. **Guldasta-e-Prefetcher (Optimized)**

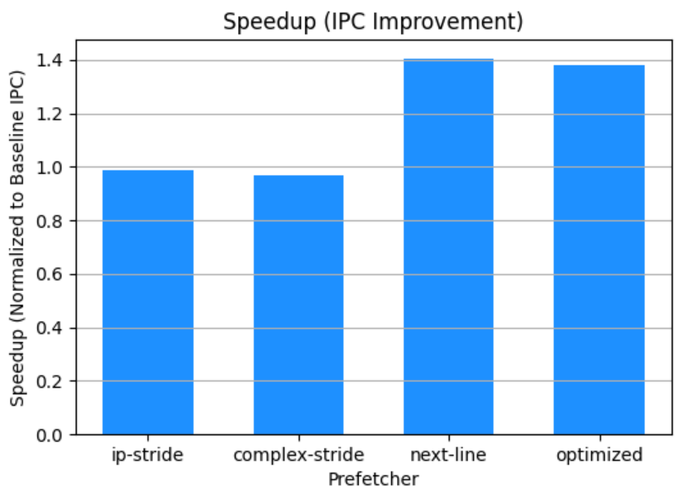
The table below summarizes the speedup observed for each prefetcher.

Table 1: Speedup Comparison for Traces 1, 2, and 3

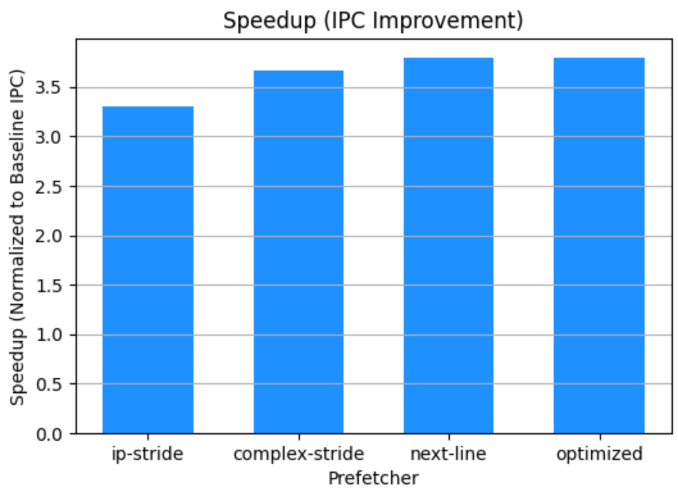
Trace	IP-Stride	Complex-Stride	Next-Line	Optimized
Trace 1	0.986	0.966	1.40	**1.38**
Trace 2	3.31	3.66	3.80	**3.79**
Trace 3	1.24	1.19	0.88	**1.19**

Figure 1: Speedup Analysis for Different Prefetchers

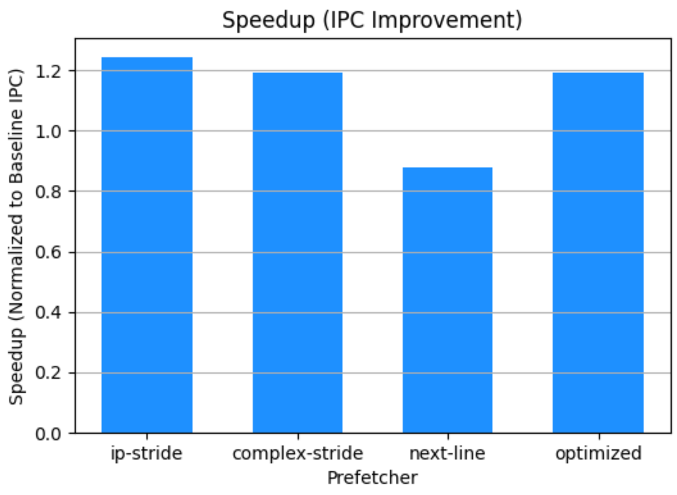
1. **Trace 1**



2. **Trace 2**



3. **Trace 3**



4.2 L1D MPKI Analysis

L1D MPKI (Misses Per Kilo Instructions) is a crucial metric to measure prefetching efficiency. The lower the MPKI, the more effective the prefetcher.

Table 2: L1D MPKI Comparison for Trace 1

Prefetcher	L1D MPKI (Load)	L1D MPKI (Total)
No Prefetcher	45.4355	48.21
IP-Stride	45.6271	49.2384
Complex-Stride	45.9344	52.2531
Next-Line	49.1029	216.727
Optimized	49.0278	215.469

Table 3: L1D MPKI Comparison for Trace 2

Prefetcher	L1D MPKI (Load)	L1D MPKI (Total)
No Prefetcher	69.9647	69.9934
IP-Stride	16.067	70.7358
Complex-Stride	10.1996	72.2836
Next-Line	11.1521	73.259
Optimized	11.1701	73.2536

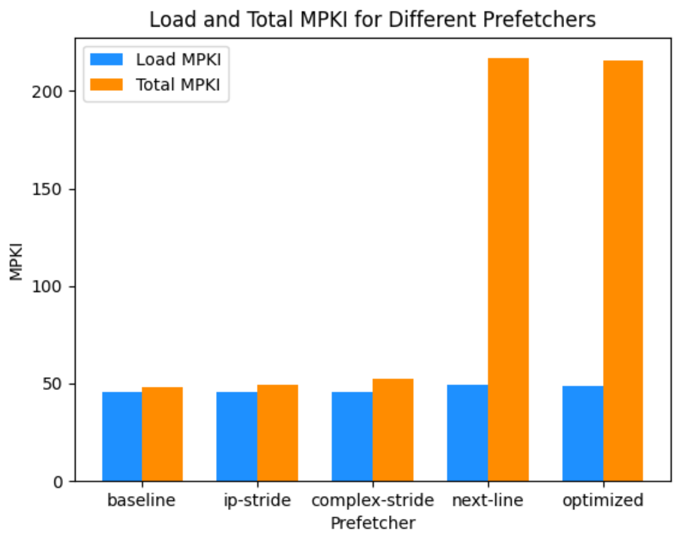
Table 4: L1D MPKI Comparison for Trace 3

Prefetcher	L1D MPKI (Load)	L1D MPKI (Total)
No Prefetcher	154.963	155.279
IP-Stride	121.588	155.447

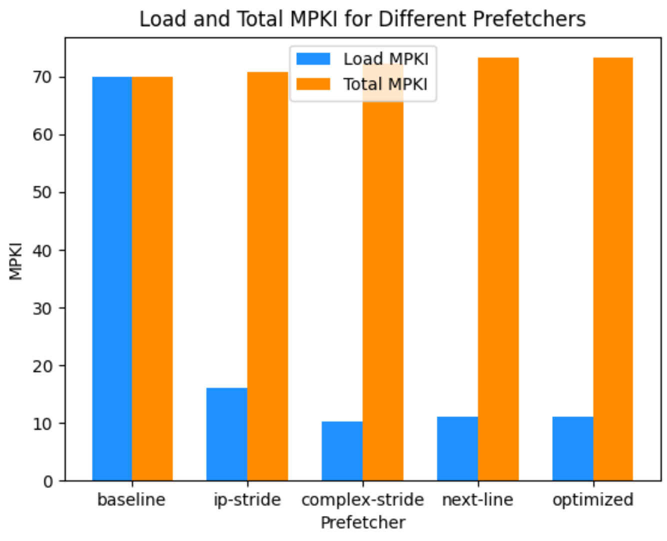
Prefetcher	L1D MPKI (Load)	L1D MPKI (Total)
Complex-Stride	116.931	166.655
Next-Line	121.98	519.913
Optimized	116.956	167.325

Figure 2: L1D MPKI Analysis for Traces 1, 2, and 3

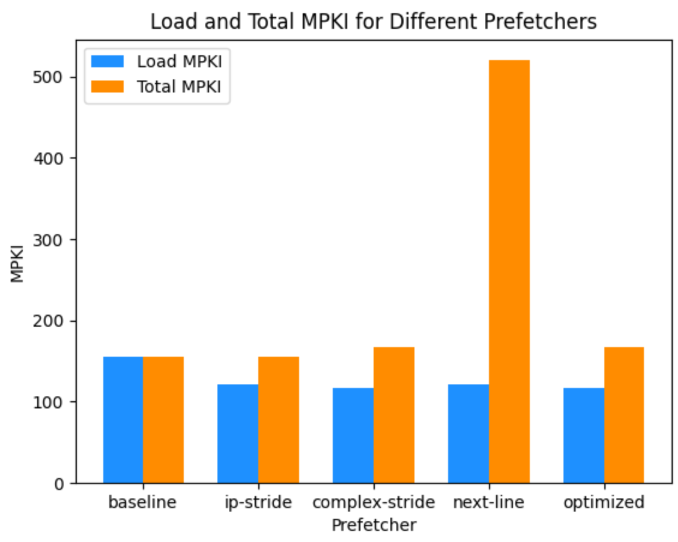
1. **Trace 1**



2. **Trace 2**



3. **Trace 3**



5. Observations and Key Takeaways

1. **IP-Stride Prefetcher**:
    - Performed well in `Trace 3`and `Trace1`, where it could capture the memory access pattern.
    - Struggled in `Trace 2` due to strides clashing for the same tracker slot and confidence.
2. **Complex-Stride Prefetcher**:
    - Showed relatively high accuracy across all traces.
3. **Next-Line Prefetcher**:
    - Effective for highly sequential workloads (`Trace 1` & `Trace 2`) but underperformed for traces with irregular patterns.
4. **Guldasta-e-Prefetcher (Optimized)**:
    - Was able to select the best (close to best for trace 3) prefetcher across all traces, demonstrating the advantage of dynamically selecting prefetcher.

6. Conclusion

The **Guldasta-e-Prefetcher** successfully combines multiple prefetchers, dynamically adapting to varying workload characteristics. This implementation significantly reduces L1D MPKI and boosts speedup compared to individual prefetchers, making it a robust solution for diverse workloads. Further refinements, such as adaptive `PHASE_LENGTH` or hybrid state machines, could yield even better results.

Task 4

Feedback-Directed Prefetching in ChampSim

- Reference: https://doi.org/10.1109/HPCA.2007.346185

1. Motivation

Hardware prefetchers are crucial for improving memory access latency by predicting and fetching data before it is requested by the processor. However, aggressive prefetching can lead to bandwidth wastage and cache pollution when inaccurate prefetches are made. Balancing the timeliness, accuracy, and pollution of prefetching is essential for performance improvement.

The goal of implementing feedback-directed prefetching is to dynamically adjust prefetch aggressiveness based on runtime characteristics like accuracy, lateness, and pollution. This approach enhances the balance between aggressive prefetching for timely access and conservative prefetching to reduce pollution, ensuring efficient use of memory bandwidth.

2. Design Overview

The feedback-directed prefetcher throttler adapts prefetching behavior by continuously monitoring metrics such as:

- **Accuracy**: The ratio of useful prefetches to total prefetches.
- **Lateness**: The fraction of useful prefetches that arrive after they are needed.
- **Pollution**: The ratio of cache pollution induced by prefetches to demand misses.

Based on these metrics, the throttler adjusts two key parameters:

- **Prefetch degree**: The number of prefetches issued for each demand miss.
- **Prefetch distance**: The distance between the demand request and the prefetch in memory address space.

3. Implementation Details

The prefetch throttler operates within a defined window of cache accesses (denoted as `thr_window_len`), updating prefetch parameters when the window is completed. The throttler logic is designed for the L1 data cache (L1D) and only activates when the system is out of warmup mode.

At each interval, the following actions take place:

1. **Metric Calculation**: The prefetch throttler tracks several metrics—prefetch accuracy, lateness, induced pollution, and coverage. These metrics are updated using exponential decay (50% weight for past intervals and 50% for the current window) to ensure the throttler adapts smoothly to changing workloads.

2. **Throttling Logic**: The prefetcher's aggressiveness is controlled by a `config_counter` that can be incremented or decremented based on the computed metrics. The following conditions are checked:

   - High accuracy and lateness lead to increasing prefetch timeliness by incrementing the `config_counter`.
   - High accuracy but excessive pollution leads to decreasing aggressiveness by decrementing the `config_counter`.
   - Medium accuracy with either high lateness or high pollution also triggers adjustments.
   - Low accuracy in combination with high lateness or pollution results in conservative prefetching.

3. **Prefetch Parameter Adjustment**: The throttler uses the value of `config_counter` to set the prefetch distance and degree. These values determine how far ahead the prefetcher looks (distance)

and how many prefetches it issues for each demand miss (degree):

- **Very conservative**: Prefetch distance of 2, prefetch degree of 1.
- **Conservative**: Prefetch distance of 4, prefetch degree of 2.
- **Middle-of-the-road**: Prefetch distance of 8, prefetch degree of 5.
- **Aggressive**: Prefetch distance of 16, prefetch degree of 6.
- **Very aggressive**: Prefetch distance of 32, prefetch degree of 8.

4. **Metrics Reset**: After every interval, all counters (`thr_pf_sent`, `thr_pf_useful`, `thr_pf_late`, `thr_demand_miss_total`, `thr_pf_pollution_total`) are reset for the next window.

4. Summary

This implementation of feedback-directed prefetching dynamically adapts prefetch aggressiveness by monitoring and adjusting prefetch parameters based on runtime metrics. The throttler ensures that the prefetcher remains aggressive when it is beneficial (high accuracy and low pollution) and conservative when prefetches are either late, inaccurate, or polluting the cache.

By adjusting prefetch behavior at runtime, this method aims to maximize the benefits of prefetching (increased cache hits, reduced memory latency) while minimizing its downsides (cache pollution and bandwidth wastage), thereby improving overall system performance.