Advanced Computer Architecture
Programming Assignment 5

Faculty of Computer Engineering

Abolfazl Bayat
Prof: Dr. Pejman Lotfi-Kamran

Des 2023
Azar 1402

Contents

1. Introduction to the problem

Overview
In this assignment, you will extend the cache simulator that you developed in PA1 to assess the effectiveness of two widely-used prefetchers for hiding instructions and data access latencies of data center and database applications using traces of CloudSuite and OLTP workloads. You are free to choose the programming language of your choice among C/C++, Java, Perl, PHP, or Python. You are advised to start early.

Programming Exercise
For this programming assignment, you implement two prefetchers: (1) next-line prefetcher, and (2) stride prefetcher. A next-line prefetcher is a simple prefetcher that after an access to a cache block that results in a miss, predicts future accesses to the next cache block, and prefetches the predicted cache block if it is not in the cache. If a predicted cache block actually being used by the processor, the behavior of a next-line prefetcher is identical to the behavior of the prefetcher in the case of a cache miss.
While a bit more complex than a next-line prefetcher, a stride prefetcher is a simple prefetcher for prefetching data references characterized by regular strides. A stride prefetcher attempts to uncover regular strides for every static load or store. For this purpose, a stride prefetcher has a reference prediction table (RPT), which is a cache tagged and referenced with the PC of load and store instructions. The entries in the RPT hold the previous address referenced by the corresponding instruction (last address), the offset of that address from the previous data address referenced by that instruction (last stride), and some flags. When a load or store instruction is executed that matches an entry in the RPT, the offset of the data address of that load or store from the previous data address stored in the RPT is calculated (current stride). When this matches the last stride, a prefetch is launched for the data address one offset ahead of the current data address (i.e., current address + stride). Moreover, on every access, the last address and last stride fields of the entry in the RPT that corresponds to the load or store instruction get updated.

Note 1: In this programming assignment, there is no restriction on the size of the reference prediction table.
Note 2: The first column of the L1d trace files contains the PCs of load and store instructions. Please notice that every load or store has a unique PC, and consequently, such PCs can be used to distinguish different load and store instructions.

Milestone 1
For every workload, measure the coverage and accuracy of the next-line prefetcher and the stride prefetcher for an L1d cache. Just like PA1, a 32-KB, 8-way associative cache with a 64B block size that uses the LRU replacement policy will be used in this programming assignment.

Milestone 2 (OPTIONAL but HIGHLY recommended)
Propose a prefetcher with a higher coverage and accuracy as compared to next-line and stride prefetchers. Implement the proposed prefetcher in your cache simulator and measure its effectiveness (what matters the most is innovation!).

Deliverable
Hand in the code and a short report (PDF) that describes the two prefetchers and what you observed in this experiment. Please include the coverage and accuracy of the two prefetchers for every workload in your report in a form of a graph (more precisely, a bar chart). Your graphs need to be readable and clear. You can look at a prefetching paper to see how such graphs should look (e.g., Figure 6 of Spatial Memory Streaming). Moreover, for every prefetcher, please add a bar to your graphs to show the average coverage and accuracy across all workloads. What did you learn from this experiment?

If you have done Milestone 2, please clearly explain the proposed prefetcher, justify why you believe it works, and report the simulation numbers to back up your claim.

Note: One of the goals of this programming assignment is for you to learn how to present experimental numbers in a form of a graph. Hence, spend as much time as necessary to make your graphs look great!

2. Set-associative cache simulator

The provided code is a cache simulator written in Python. It simulates the behavior of a cache memory system with different replacement policies using a set-associative cache. The supported replacement policies are Least Recently Used (LRU), Not Most Recently Used (NMRU), and RANDOM. The simulator reads memory addresses from benchmark files and simulates cache hits and misses based on the specified cache configuration and replacement policy. It then measures and prints the hit ratio for each benchmark and replacement policy.

Overview of Key Functions:

1. **convert_to_bytes(size):**
- Converts a user-specified size (in KB, MB, GB) to bytes.
- Handles different units such as B, KB, MB, GB.

2. **clear_cache(number_of_sets):**
- Initializes an empty cache with the specified number of sets.

3. **replacement(policy, cache, mm_block_number, set_number):**
- Implements the replacement policy for a cache set when a miss occurs.
- Supports LRU, NMRU, and RANDOM replacement policies.

4. **measure_cache(policy, hit_count, miss_count):**
- Computes and prints the hit ratio based on the number of hits and misses.
- Prints a separator line for clarity, particularly after LRU policy.

5. **get_cache_specification_by_user():**
- Prompts the user to input cache specifications such as size, block size, word size, and associativity.
- Provides default values if the user does not specify.

6. **get_benchmarks():**
- Constructs the path to the benchmark files (traces) to be used in simulations.

7. **lookup(cache, mm_block_number, set_number, policy, hit_count, miss_count):**
- Simulates a memory address lookup in the cache.
- Updates hit and miss counts accordingly.
- Implements cache hit and miss policies.

8. **main():**
- Serves as the main entry point.
- Iterates over benchmark files, applying specified replacement policies and measuring hit ratios for each benchmark.
- Outputs results for each benchmark and replacement policy.

Key Concepts:

- The cache is organized in sets, and each set contains a specified number of cache lines.
- Memory addresses are mapped to cache sets and blocks based on the specified cache configuration.
- Replacement policies determine which cache line to replace when a miss occurs.

2.1 Implementation of set-associative cache simulator

In this section I provide a detailed explanation of this code line by line.

```
import os
from collections import OrderedDict
import random
```

Import Statements:

- Import necessary modules (os for operating system-specific functions, OrderedDict for an ordered dictionary, and random for random number generation).

```
def convert_to_bytes(size):
    if not size:
        return 0
    # Split the input string into numeric part and unit
    x_str = ''.join(filter(str.isdigit, size))
    x = float(x_str) if x_str else 0  # Convert numeric part to
    float, default to 0 if not present

    y = ''.join(filter(str.isalpha, size)).upper()
    # Convert based on the unit
    if y == "B":
        return x
    elif y == "KB":
        return x * 1024
    elif y == "MB":
        return x * 1024 * 1024
    elif y == "GB":
        return x * 1024 * 1024 * 1024
    else:
        raise ValueError("Invalid unit: {}".format(y))
```

convert_to_bytes(size):

- Converts a user-specified size (in KB, MB, GB) to bytes.
- Extracts numeric and alphabetic parts of the input string to determine the size and unit.
- Converts the numeric part to bytes based on the specified unit.
- Handles different units (B, KB, MB, GB) and raises an error for an invalid unit.

```
def clear_cache(number_of_sets):
    cache = OrderedDict()
    for i in range(number_of_sets):
        cache[f"s{i}"] = []
    return cache
```

clear_cache(number_of_sets):

- Initializes an empty cache with a specified number of sets using an ordered dictionary.
- Sets are labeled as "s0", "s1", ..., "sn" where n is the number of sets.

```
def replacement(policy, cache, mm_block_number, set_number):
    if policy == "LRU":
        cache[f"s{set_number}"] = cache[f"s{set_number}"][1:]
        cache[f"s{set_number}"].append(mm_block_number)
    elif policy == "NMRU":
        random_num = random.randint(0,6)
        cache[f"s{set_number}"].pop(random_num)
        cache[f"s{set_number}"].append(mm_block_number)
    else:
        random_num = random.randint(0,7)
        cache[f"s{set_number}"].pop(random_num)
        cache[f"s{set_number}"].append(mm_block_number)
```

replacement(policy, cache, mm_block_number, set_number):

- Implements the replacement policy for a cache set when a miss occurs.
- Supports three replacement policies: LRU, NMRU, and RANDOM.
    - For LRU, it moves the least recently used element to the end.
    - For NMRU and RANDOM, it randomly selects an element to replace.

```
def measure_cache(policy, hit_count, miss_count):
   hit_ratio = round((hit_count / (hit_count + miss_count)), 8)
   * 100
   print(f"Hit Ratio = {hit_ratio}%")
   if policy == "LRU": print("--------------------------------")
```

measure_cache(policy, hit_count, miss_count):

- Computes and prints the hit ratio based on the number of hits and misses.
- Rounds the hit ratio to 8 decimal places and prints the result as a percentage.
- If the policy is LRU, it prints a separator line for clarity.

```
def get_cache_specification_by_user():
   ''' declare specification of cache by user'''
   cache_size = convert_to_bytes(input("(enter the Cache size
   - use KB, MB, GB ... - or press ENTER for default (32KB)\ncache
   size : "))
   block_size = convert_to_bytes(input("(enter the Block size - use
   B, KB, ... - or press ENTER for default (64B)\nBlock size : "))
   word_size = convert_to_bytes(input("(enter the Word size
   - use B, ... - or press ENTER for default (1B)\nWord size : "))
```

```
        way = input("enter the associativity of cache - x-way
        set-associative - or press ENTER for default (8)\nx: ")

        if cache_size == 0: cache_size = 32*1024 # 32KB
        if word_size == 0: word_size = 1
        way = 8 if way == "" else int(way)
        block_size = int(64 / word_size) if block_size == 0
        else int(block_size / word_size)

        number_of_sets = int(((cache_size / block_size)) / way)

        cache = OrderedDict()
        for i in range(number_of_sets):
            cache[f"s{i}"] = []

        return [cache, block_size, number_of_sets]
```

get_cache_specification_by_user():

- Prompts the user to input cache specifications (size, block size, word size, and associativity).
- Provides default values if the user does not specify.
- Calculates the number of sets based on the cache size, block size, and associativity.
- Initializes an ordered dictionary for the cache.

```
    def get_benchmarks():
        current_directory = os.getcwd() # fetch the benchmark path
        relative_path = 'traces/'
        main_directory = os.path.join(current_directory, relative_path)
        return main_directory
```

get_benchmarks():

- Retrieves the path to the benchmark files (traces/) by combining the current directory with the relative path.

```
def lookup(cache, mm_block_number, set_number, policy,
hit_count, miss_count):
    ''' block in cache checking...'''
    if mm_block_number in cache[f"s{set_number}"]:
        ''' hit policy '''
        hit_count+=1
        current_block =
        cache[f"s{set_number}"].index(mm_block_number)
        element_to_move = cache[f"s{set_number}"].pop(current_block)
        cache[f"s{set_number}"].append(element_to_move)
    else:
        ''' miss policy '''
        miss_count+=1
        ''' checking if cache is full...'''
        if len(cache[f"s{set_number}"]) < 8:
            cache[f"s{set_number}"].append(mm_block_number)
        else:
            replacement(policy, cache, mm_block_number, set_number)
    return [cache, hit_count, miss_count]
```

lookup(cache, mm_block_number, set_number, policy, hit_count, miss_count):

- Simulates a memory address lookup in the cache.
- Updates hit and miss counts based on the cache hit or miss.
- Implements cache hit and miss policies.
- If it's a hit, it moves the accessed block to the end to represent its recent use. If it's a miss, it checks if the cache is full and performs replacement accordingly.

```
def main():
    cache, block_size, number_of_sets =
    get_cache_specification_by_user()

    # replacement_policies = ["LRU", "NMRU", "RANDOM"]
    replacement_policies = ["LRU"]
    ''' measure hit ratio of cache with some workloads '''
    print("---------------------------------")
    print("The test operation has started")
    print("--------------------------------")

    main_directory = get_benchmarks()

    ''' Iterate through each subdirectory in the main directory
    (benchmarks paths) '''
    for subdir, dirs, files in os.walk(main_directory):
        for benchmark in files:
            benchmark_path = os.path.join(subdir, benchmark)
            bechmark_name = (subdir.split("/"))[-1]
            print(f"On {bechmark_name} Benchmark")
            for policy in replacement_policies:
                print(f"with {policy} policy", end=" ")
                with open(benchmark_path) as file:

                    ''' reset the cache '''
                    cache = clear_cache(number_of_sets)
                    hit_count = miss_count = 0

                    for line in file: # read addresses line by line
                        pc_addr = line.split()
                        address = int(pc_addr[1], 16)
                        mm_block_number = address // block_size
                        set_number = mm_block_number % number_of_sets

                        cache, hit_count, miss_count = lookup(cache,
                        mm_block_number, set_number, policy,
                        hit_count, miss_count)

                measure_cache(policy, hit_count, miss_count)
    print("The test operation is over")
    print("--------------------------------")

if __name__ == "__main__":
    main()
```

main():

- Initializes the cache, block size, and number of sets based on user specifications.
- Specifies the replacement policies to be used (in this case, only LRU is included).

- Iterates through each benchmark file in the specified directory.
- For each benchmark, it iterates through the specified replacement policies.
- Reads memory addresses from the benchmark file, simulates cache operations, and measures hit ratios.
- Prints results for each benchmark and replacement policy.

The script uses the specified replacement policy (in this case, LRU) for each benchmark. It reads the memory addresses from benchmark files, simulates cache hits and misses, and measures the hit ratio for each replacement policy and benchmark.

3. Cache Prefetching

Cache prefetching is a technique used in computer systems to improve memory access latency by fetching data from main memory into the cache before it is actually requested by the processor. The goal is to anticipate future memory accesses and proactively bring the required data into the cache to reduce the likelihood of cache misses. This helps in hiding memory access latencies and can lead to better overall system performance.

Prefetching mechanisms aim to exploit spatial and temporal locality in program behavior. Spatial locality refers to the tendency of a program to access nearby memory locations, and temporal locality refers to the tendency to access the same memory locations over a short period. By prefetching data based on these patterns, cache prefetchers can enhance the efficiency of the memory hierarchy.

Types of Cache Prefetchers:

1. **Next-Line Prefetcher:**

- The next-line prefetcher predicts that the next cache line will be accessed after the current line. It proactively fetches the next cache line into the cache, assuming a sequential memory access pattern.

2. **Stride Prefetcher:**

- The stride prefetcher predicts memory accesses based on a regular stride or pattern. It observes a constant difference (stride) between consecutive memory addresses and fetches data accordingly. This is useful for programs with regular access patterns.

Implementation of Cache Prefetchers:
Now, let's extend the provided code to include a Next-Line Prefetcher and a Stride Prefetcher. The prefetchers will be incorporated into the existing cache simulation framework to observe their impact on cache performance.

3.1 Next-line Prefetcher

The extended code includes a Next-Line Prefetcher to the existing cache simulator. This prefetcher predicts that the next cache line will be accessed after the current line and proactively fetches the next cache line into the cache. The code structure is similar to the original cache simulator, with added functionality for prefetching.

Changes Compared to the Previous Code:

1. **Prefetch Function:**
- Introduced a new function named prefetch(cache, pred_mm_block_number, pred_set_number, policy) to simulate prefetching.
- Checks if the predicted block is already in the cache. If yes, applies hit policy; otherwise, apply miss policy.
- If the cache is full, it performs replacement using the specified policy.

2. **Lookup Function Changes:**
- Modified the existing lookup function to return a missed flag indicating whether a cache miss occurred.
- The missed flag is used to trigger the prefetching logic.

3. **Main Function Changes:**
- Added logic to handle prefetching in the main loop.
- Introduced a variable pred_mm_block_number to track the predicted next block for prefetching.
- If a cache miss occurs, the next block is predicted, and the prefetch function is called to bring it into the cache.

4. **Prefetching Conditions:**
- Prefetching is triggered when a cache miss occurs.
- The next block is predicted to be the current block's successor (pred_mm_block_number = mm_block_number + 1).

- Prefetching is also triggered when the current block matches the predicted block, ensuring continuous prefetching in case of sequential access.

Key Concepts:

- The next-line prefetcher aims to proactively bring the next cache line into the cache, anticipating sequential memory accesses.
- It leverages the existing cache simulator framework, extending it to include prefetching functionality.
- The prefetching logic is based on the occurrence of cache misses, and it aims to improve overall cache performance by reducing latency.

3.1.1 Implementation of Next-line Prefetcher

In this section I provide a detailed explanation of this code line by line.

```
def prefetch(cache, pred_mm_block_number, pred_set_number, policy):
    ''' block in cache checking...'''
    if pred_mm_block_number in cache[f"s{pred_set_number}"]:
        ''' hit policy '''
        current_block =
        cache[f"s{pred_set_number}"].index(pred_mm_block_number)
        element_to_move =
        cache[f"s{pred_set_number}"].pop(current_block)
        cache[f"s{pred_set_number}"].append(element_to_move)
    else:
        ''' miss policy '''
        ''' checking if cache is full...'''
        if len(cache[f"s{pred_set_number}"]) < 8:
            cache[f"s{pred_set_number}"].append(pred_mm_block_number)
        else:
            replacement(policy, cache, pred_mm_block_number,
            pred_set_number)
    return [cache]
```

prefetch(cache, pred_mm_block_number, pred_set_number, policy):

- This function simulates the prefetching mechanism.
- It checks if the predicted block is already in the cache. If yes, it applies the hit policy.
- If the block is not in the cache (cache miss), it applies the miss policy.
- If the cache is full, it performs replacement using the specified policy.

```
def lookup(cache, mm_block_number, set_number, policy, hit_count,
miss_count):
    missed = 0
    ''' block in cache checking...'''
    if mm_block_number in cache[f"s{set_number}"]:
        ''' hit policy '''
        hit_count+=1
        current_block =
        cache[f"s{set_number}"].index(mm_block_number)
        element_to_move = cache[f"s{set_number}"].pop(current_block)
        cache[f"s{set_number}"].append(element_to_move)
    else:
        ''' miss policy '''
        missed = 1
        miss_count+=1
        ''' checking if cache is full...'''
        if len(cache[f"s{set_number}"]) < 8:
            cache[f"s{set_number}"].append(mm_block_number)
        else:
            replacement(policy, cache, mm_block_number, set_number)
    return [cache, hit_count, miss_count, missed]
```

lookup(cache, mm_block_number, set_number, policy, hit_count, miss_count):

- Modified the existing lookup function to return a missed flag indicating whether a cache miss occurred.
- If a cache miss occurs, the missed flag is set to 1, triggering the prefetching logic in the main loop.

```
for line in file: # read addresses line by line
    pc_addr = line.split()
```

```
        address = int(pc_addr[1], 16)
        mm_block_number = address // block_size
        set_number = mm_block_number % number_of_sets
        cache, hit_count, miss_count, missed = lookup(cache,
        mm_block_number, set_number, policy, hit_count, miss_count)

          if missed:
               pred_mm_block_number = mm_block_number + 1
               pred_set_number = pred_mm_block_number % number_of_sets
               prefetch(cache, pred_mm_block_number, pred_set_number,
               policy)
          elif mm_block_number == pred_mm_block_number:
               pred_mm_block_number = mm_block_number + 1
               pred_set_number = pred_mm_block_number % number_of_sets
               prefetch(cache, pred_mm_block_number, pred_set_number,
               policy)
```

Main Loop:

- In the main loop where memory addresses are processed, after a cache miss, the missed flag is checked.
- If a miss occurs, the pred_mm_block_number is predicted to be the next block (mm_block_number + 1).
- The prefetch function is then called to bring the predicted block into the cache.
- Additionally, if the current block matches the predicted block, prefetching is continued, ensuring a continuous prefetching process for sequential access patterns.

These changes enhance the existing cache simulator code to incorporate next-line prefetching logic.

3.2 Stride Prefetcher

The provided code extends the existing cache simulator to incorporate a Stride-Prefetcher. The Stride-Prefetcher predicts memory accesses based on a regular stride or pattern and proactively fetches data into the cache. This type of prefetcher is effective when the program exhibits a predictable stride in its memory access pattern.

Changes Compared to the Previous Code:

1. **RPT Data Structure:**

- Introduced the Relative Page Table (RPT) data structure to store information about the program's memory access pattern.
- RPT is a list of lists, where each inner list represents a program counter (PC), the last accessed memory address, and the stride between consecutive memory accesses for that PC.

2. **prefetch Function Changes:**

- Modified the prefetch function to return the updated cache after performing prefetching.
- The function now takes the predicted next block, predicted set number, and the cache policy as parameters.
- The Stride-Prefetcher logic is integrated into this function to handle both hit and miss policies.

3. **Main Loop Changes:**

- Added logic for Stride-Prefetching within the main loop that processes memory addresses.
- For each memory access, the code checks if the program counter (PC) is present in the RPT.
- If present, it calculates the current stride and predicts the next address based on the stride.
- If the stride remains the same, the next address is predicted, and prefetching is performed.
- If the PC is not in the RPT, a new entry is added to the RPT.

Key Concepts:

- The Stride-Prefetcher leverages the RPT data structure to track the memory access patterns for different program counters.
- For each memory access, the code checks the RPT to determine if there's a predictable stride.
- If a predictable stride is found, the next memory address is predicted, and prefetching is performed.
- The Stride-Prefetcher aims to enhance cache performance by proactively fetching data based on regular memory access patterns.

3.2.1 Implementation of Stride Prefetcher

In this section I provide a detailed explanation of this code line by line.

```
def prefetch(cache, pred_next_block, pred_set_number, policy):
    ''' block in cache checking...'''
    if pred_next_block in cache[f"s{pred_set_number}"]:
        ''' hit policy '''
        current_block =
        cache[f"s{pred_set_number}"].index(pred_next_block)
        element_to_move =
        cache[f"s{pred_set_number}"].pop(current_block)
        cache[f"s{pred_set_number}"].append(element_to_move)
    else:
        ''' miss policy '''
        ''' checking if cache is full...'''
        if len(cache[f"s{pred_set_number}"]) < 8:
            cache[f"s{pred_set_number}"].append(pred_next_block)
        else:
            replacement(policy, cache, pred_next_block,
            pred_set_number)
    return cache
```

prefetch(cache, pred_next_block, pred_set_number, policy):

- This function simulates the prefetching mechanism for the Stride-Prefetcher.
- It checks if the predicted next block is already in the cache. If yes, it applies the hit policy.
- If the block is not in the cache (cache miss), it applies the miss policy.
- If the cache is full, it performs replacement using the specified policy.
- The function returns the updated cache after prefetching.

```
for line in file: # read addresses line by line
      is_pc_in_RPT = 0
      pc_addr = line.split()
      address = int(pc_addr[1], 16)
```

```
pc = int(pc_addr[0], 16)
mm_block_number = address // block_size
set_number = mm_block_number % number_of_sets
cache, hit_count, miss_count = lookup(cache, mm_block_number,
set_number, policy, hit_count, miss_count)
''' stride prefetching...'''
for i in range(len(RPT)):
    this_pc, last_address, last_stride = RPT[i]
    # print(RPT[i])
    if this_pc == pc:
        is_pc_in_RPT = 1
        current_stride = address - last_address
        if last_stride == current_stride:
            next_address = address + last_stride
            pred_next_block = next_address // block_size
            pred_set_number = pred_next_block %
            number_of_sets
            cache = prefetch(cache, pred_next_block,
            pred_set_number, policy)
            break
        else:
            # update_RPT
            RPT[i] = [pc, address, current_stride]
            break
if not is_pc_in_RPT:
    RPT.append([pc, address, 0])
```

Main Loop:

- In the main loop where memory addresses are processed, the code checks if the program counter (PC) is present in the RPT.
- If the PC is in the RPT, it calculates the current stride between the current and last memory addresses.
- If the stride remains the same, it predicts the next address and performs prefetching using the prefetch function.
- If the PC is not in the RPT, a new entry is added to the RPT.

Key Concepts:

- The main loop iterates through memory addresses, and for each address, it checks if there's a predictable stride in the Stride-Prefetcher's RPT.

- If a predictable stride is found, the next address is predicted, and prefetching is performed.
- The RPT stores information about the program counter, the last accessed memory address, and the stride between consecutive accesses.
- The Stride-Prefetcher aims to improve cache performance by anticipating regular patterns in memory accesses.

These changes enhance the existing cache simulator code to include the Stride-Prefetcher logic.

3.3 Other Approaches

In addition to the Next-line Prefetcher and Stride Prefetcher, several other prefetching strategies were explored to further enhance hit ratios. Here are brief explanations of each method:

Next-2Line Prefetcher:
This prefetcher, an extension of the Next-line Prefetcher, predicts and prefetches the next two cache lines instead of just one. This method aims to capture more locality in memory accesses.

Next-3Line Prefetcher:
Similar to the Next-2Line Prefetcher, this approach predicts and prefetches the next three cache lines.

Combine of Next-Line and Stride Prefetchers:
This approach combines the strengths of the Next-line and Stride prefetchers. The Next-line prefetcher focuses on sequential accesses, while the Stride prefetcher identifies and prefetches based on regular stride patterns. Combining these methods aims to cover a broader range of memory access patterns.

Combine of Next-2Line and Stride Prefetchers:
Extending the combination strategy, this approach integrates the Next-2Line Prefetcher with the Stride Prefetcher.

Combine of Next-3Line and Stride Prefetchers:
Similar to the previous method, this approach combines the Next-3Line Prefetcher with the Stride Prefetcher.

Comparison of Approaches

Now, let's compare the hit ratios achieved by each prefetching strategy across various benchmarks. The results are shown at table1 of the next section for the set-associative cache with LRU replacement policy.

4. Evaluation

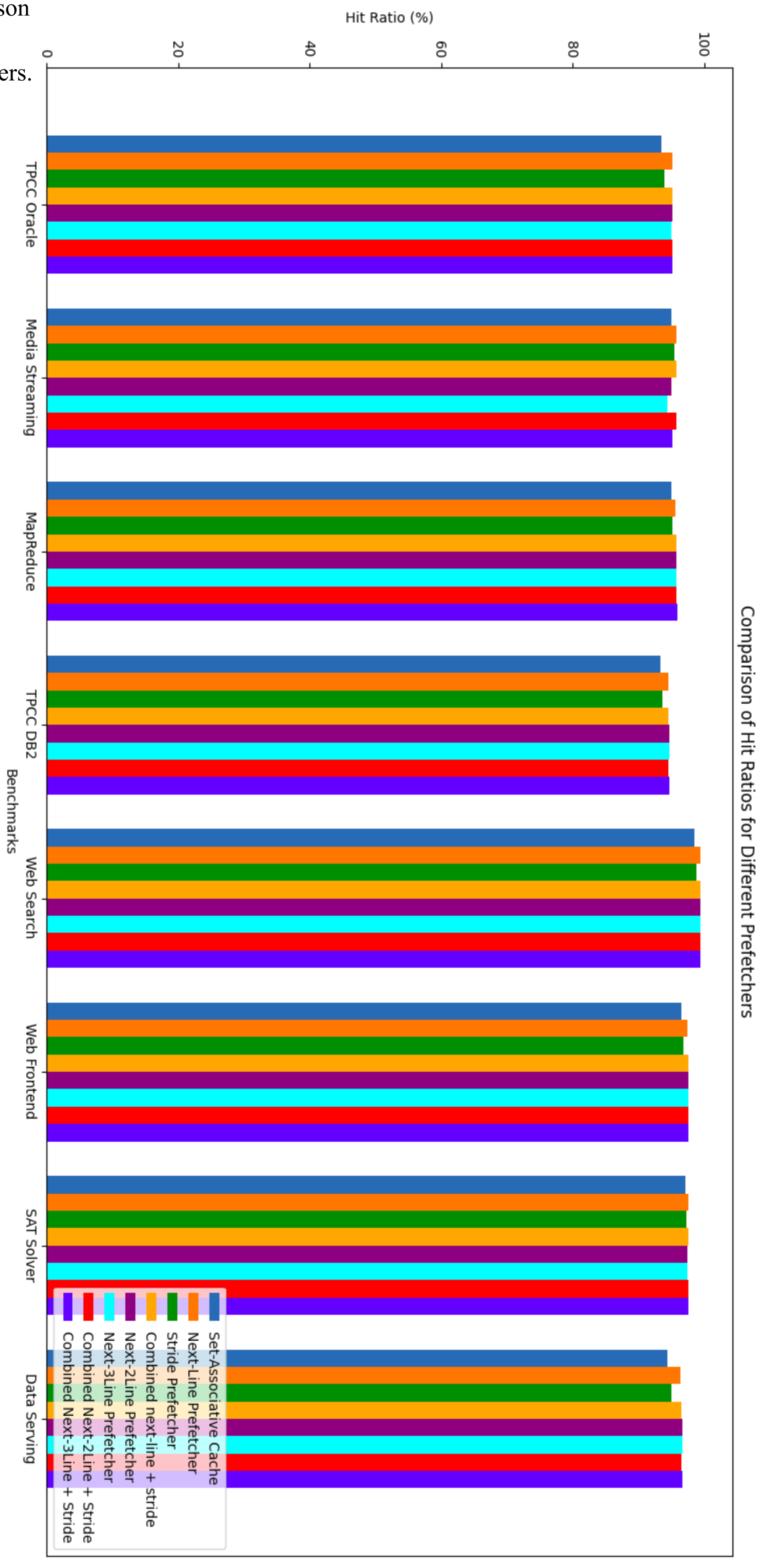
In this evaluation, seven different cache configurations were tested on various benchmarks, with the set-associative cache serving as the base code and baseline hit ratios. The seven additional configurations were implemented to enhance the hit ratios compared to the baseline. Below are the detailed results for each configuration in benchmarks (traces10M):

Table 1: Comparison of Hit Ratios for Different Prefetching Strategies in Set-Associative Cache

Benchmarks	**[1]**[1]	**[2]**[2]	[3]**[3]**	[4]**[4]**	[5]**[5]**	[6]**[6]**	[7]**[7]**	[8]**[8]**
TPCC Oracle	93.40	94.96	93.80	95.01	94.95	94.85	94.98	94.88
Media Streaming	94.81	95.59	95.37	95.60	94.93	94.26	94.95	94.27
MapReduce	94.82	95.53	94.97	95.62	95.69	95.69	95.77	95.77
TPCC DB2	93.26	94.41	93.53	94.47	94.57	94.60	94.62	94.64
Web Search	98.31	99.28	98.63	99.33	99.31	99.30	99.34	99.33
Web Frontend	96.43	97.37	96.75	97.41	97.45	97.46	97.49	97.46
SAT Solver	96.93	97.38	97.18	97.53	97.35	97.29	97.49	97.43
Data Serving	94.25	96.22	94.90	96.35	96.56	96.60	96.60	96.66

[1] Set associative hit ratios (baseline)
[2] Next line prefetcher hit ratios
[3] Stride prefetcher hit ratios
[4] next line + stride hit ratios
[5] Next-2line prefetcher hit ratios
[6] Next-3line prefetcher hit ratios
[7] Next-2line + stride hit ratios
[8] Next_3line + stride hit ratios

Figure1: Comparison of Hit Ratios for Different Prefetchers.



5. Conclusion

The study of prefetching strategies has shown that a combination of different approaches can lead to improved hit ratios across diverse benchmarks. The effectiveness of each strategy depends on the characteristics of the workload. In real-world scenarios, selecting or combining prefetchers based on the specific memory access patterns of an application can significantly enhance cache performance and, consequently, overall system performance.

If you need the codes for these prefetcher implementations, feel free to email me at [abolfazl.byte@gmail.com].