Advanced Computer Architecture
Programming Assignment 3

Faculty of Computer Engineering

Abolfazl Bayat
Prof: Dr. Pejman Lotfi-Kamran

Nov 2023
Aban 1402

Contents

Introduction to the problem.....2
Set-associative cache simulator.....3
Implementation.....5
Evaluation & Conclusion.....9

Introduction to the problem

Overview
In this assignment, you will write a cache simulator and evaluate the LRU, NMRU, and Random replacement policies using traces of CloudSuite and OLTP workloads. You are free to choose the programming language of your choice among C/C++, Java, Python, or Perl. You are advised to start early.

Programming Exercise
Code a generic cache component which is parametrized by (1) Cache Block Size, (2) Cache Associativity, and (3) Cache Size. The cache component, among other things, should have a `lookup` method which gets an address and searches the cache to find the address. If the address is in the cache, this lookup is a hit. If the address is not in the cache, it is a miss. In this case, the block that contains the address should be inserted into the cache. If all the frames in the set that the block address maps to are occupied, the cache should pick a victim according to the replacement policy, evict the victim block, and insert the block that holds the address. Write an interface that reads addresses from the input trace files and apply them to the cache using the `lookup` method.

Note 1: You can find the format of the trace files in the 'readme.txt' file which is delivered to you along with the trace files (in 'traces/readme.txt').
Note 2: Please ignore the PC field of L1d trace files in this programming assignment.

Milestone
For every workload, compare the LRU replacement policy against NMRU and Random replacement policies for a 32KB, 8-way associative L1d cache with a 64B block size.

Deliverable
Hand in the code and a short report (PDF) that describes the three replacement policies and their observed performance (i.e., hit ratio). Please not only include performance numbers per benchmark in your report but also aggregate the performance numbers of each replacement policy. Please spend time to make your graph look great. Rank the replacement policies based on their performance. How much better is the best replacement policy as compared to the others?

Set-associative cache simulator

This code defines a cache simulator and uses it to simulate cache lookups on some specific benchmarks for different replacement policies. This code generally includes the following:

Cache Class:

- The Cache class is defined to simulate a cache with a specified block size, associativity, cache size, and replacement policy (LRU, NMRU &Random).
- It uses an `OrderedDict` to keep track of the cache content, where the keys are block addresses.
- The lookup method checks if a given memory address is present in the cache (cache hit) or not (cache miss). If it's a miss, it calls the `replace_block` method to handle the replacement according to the specified policy.
- The `replace_block` method removes the least recently used, not most recently used, or a randomly selected block based on the replacement policy and inserts the new block.

Replacement Policies:

- The code supports three replacement policies: LRU (Least Recently Used), NMRU (Not Most Recently Used), and Random.

Benchmark Simulation:

- The code then goes through a set of benchmark traces (memory access patterns) located in the specified directory.
- For each benchmark, it iterates through each replacement policy and simulates cache lookups using the addresses from the trace file.
- After simulating cache lookups for each policy, it calculates and prints the hit ratio for that policy on the current benchmark.

Print Statements:

- The code includes print statements to provide information about the benchmark being processed, the replacement policy being used, and the hit ratio achieved.

Random Policy Note:

- There's a special print statement for the "Random" replacement policy, which adds a separator line after printing the hit ratio. This is likely for better readability in the output.

Benchmark Paths:

- The code identifies benchmark paths by walking through subdirectories in the specified main directory.

Hit Ratio Calculation:

- The hit ratio is calculated for each policy by dividing the number of cache hits by the total number of cache accesses (hits + misses). The result is multiplied by 100 to get a percentage.

Cache Clearing:

- After simulating cache lookups for a particular policy on a benchmark, the cache is cleared before moving on to the next policy.
- In summary, the code simulates the behavior of a cache on different benchmarks using different replacement policies and prints the hit ratio achieved by each policy on each benchmark.

Implementation

In this section we coded this assignment and explain line by line:

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

- These lines import necessary modules: `OrderedDict` for an ordered dictionary, random for generating random numbers, and os for interacting with the operating system.

```
class Cache:
    def __init__(self, block_size, associativity, cache_size,
        replacement_policy):
        self.block_size = block_size
        self.associativity = associativity
        self.cache_size = cache_size
        self.replacement_policy = replacement_policy
        self.cache = OrderedDict()
```

- This defines a class Cache representing a cache. The __init__ method initializes the cache with specified parameters (block size, associativity, cache size, and replacement policy). It uses an OrderedDict called cache to store the contents.

```
def lookup(self, address):
        block_address = address // self.block_size

        if block_address in self.cache:
            # Cache hit
            self.cache.move_to_end(block_address)
            return True
        else:
            # Cache miss
            if len(self.cache) >= self.cache_size:
```

```
                # Cache is full, perform replacement
                self.replace_block(block_address)
            else:
                # Cache is not full, insert the block
                self.cache[block_address] = True

            return False
```

- The lookup method checks if a given memory address is in the cache. If it is a hit, it moves the accessed block to the end (most recently used). If it is a miss, it checks if the cache is full. If it is, it replaces a block using the replace_block method; otherwise, it inserts the new block.

```
def replace_block(self, block_address):
    if self.replacement_policy == "LRU":
        # Least Recently Used replacement policy
        self.cache.popitem(last=False)
    elif self.replacement_policy == "NMRU":
        # Not Most Recently Used replacement policy
        self.cache.popitem(last=True)
    elif self.replacement_policy == "Random":
        # Random replacement policy
        random_key = random.choice(list(self.cache.keys()))
        self.cache.pop(random_key)

    # Insert the new block
    self.cache[block_address] = True
```

- The replace_block method implements the replacement policy. Depending on the policy, it removes the least recently used, not most recently used, or a randomly chosen block from the cache. Then, it inserts the new block.

```
  def clear_cache(self):
      self.cache.clear()
```

- The clear_cache method clears the cache.

```
replacement_policies = ['LRU', 'NMRU', "Random"]
```

- This is a list of replacement policies that will be used during the simulation.

```
# TODO Get benchmark path

# Get the current working directory
current_directory = os.getcwd()

# Specify the relative path using './'
relative_path = 'traces/'

# Construct the full path using os.path.join
main_directory = os.path.join(current_directory, relative_path)
```

- These lines determine the path to the benchmark traces directory. It constructs the full path by joining the current working directory with the relative path "traces/".

```
# Iterate through each subdirectory in the main directory
 (benchmarks paths)
for subdir, dirs, files in os.walk(main_directory):
    # Iterate through each benchmark
    for benchmark in files:
        hit_count = 0
        miss_count = 0
        # Construct the full path to the benchmark
        benchmark_path = os.path.join(subdir, benchmark)
        bechmark_name = (subdir.split("/"))[-1]
        print(f"On {bechmark_name} Benchmark")
```

- These lines iterate through each subdirectory in the main directory, and for each subdirectory, it iterates through each benchmark file. It prints the name of the current benchmark.

```
for policy in replacement_policies:
    # Simulate cache lookups using addresses from the trace file
    cache = Cache(block_size=64, associativity=8,
    cache_size=32 * 1024 // 64, replacement_policy=policy)
            print(f"Replacement Policy is: {policy}" )
            with open(benchmark_path) as file:
                for line in file:
                    pc_addr = line.split()
                    access = cache.lookup(int(pc_addr[1], 16))
                    if access:
                        hit_count += 1
                    else:
                        miss_count += 1

            tmp_hit_ratio = round(hit_count / (hit_count +
            miss_count), 3)
            hit_ratio = tmp_hit_ratio * 100
            cache.clear_cache()
            print(f"Hit Ratio = {hit_ratio}%")
            if policy == "Random":
print("---------------------------------")
```

- For each benchmark and each replacement policy, it simulates cache lookups using addresses from the trace file. It prints the replacement policy being used. After simulating cache lookups for each policy, it calculates and prints the hit ratio for that policy on the current benchmark. The cache is cleared after each policy is simulated. If the policy is "Random," it adds a separator line in the output.

Evaluation & Conclusion

This data represents cache hit ratios for different benchmarks using three replacement policies: LRU (Least Recently Used), NMRU (Not Most Recently Used), and Random. Let's evaluate and draw conclusions from the data:

TPCC Oracle Benchmark:
LRU: 93.6%
NMRU: 63.2%
Random: 72.9%

- LRU performs exceptionally well with a high hit ratio, while Random outperforms NMRU.

Media Streaming Benchmark:
LRU: 96.5%
NMRU: 75.1%
Random: 81.5%

- LRU has a high hit ratio, and Random performs the best among the policies.

MapReduce Benchmark:
LRU: 94.8%
NMRU: 62.3%
Random: 72.9%

- LRU performs well, and Random is again more effective than NMRU.

TPCC DB2 Benchmark:
LRU: 93.4%
NMRU: 64.3%
Random: 73.7%

- LRU is strong, and Random is consistently better than NMRU.

Web Search Benchmark:
LRU: 98.3%
NMRU: 66.2%
Random: 76.8%

- LRU achieves an exceptionally high hit ratio, and Random is better than NMRU.

Web Frontend Benchmark:
LRU: 96.5%
NMRU: 70.5%
Random: 79.0%

- LRU is strong, and Random is more effective than NMRU.

SAT Solver Benchmark:
LRU: 97.0%
NMRU: 61.3%
Random: 73.1%

- LRU has a high hit ratio, and Random performs better than NMRU.

Data Serving Benchmark:
LRU: 94.3%
NMRU: 68.2%
Random: 76.7%

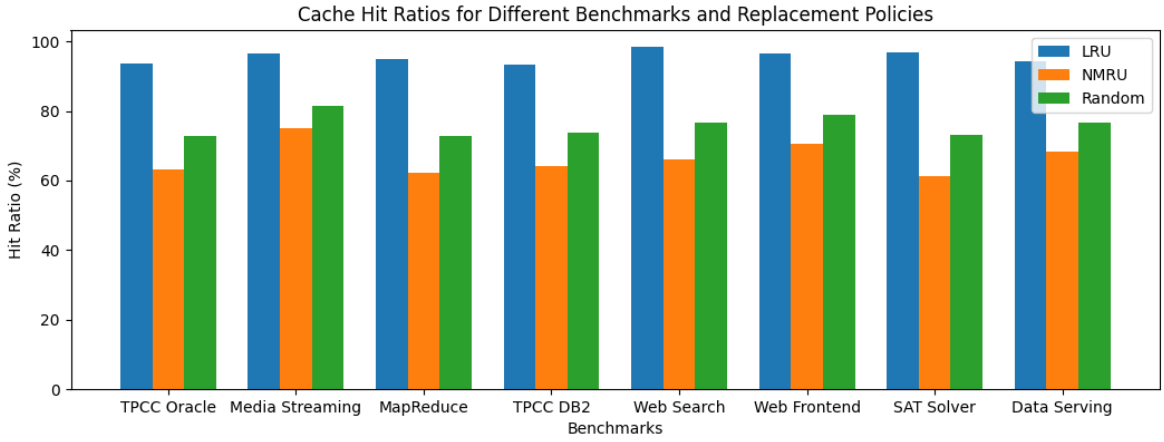
- LRU performs well, and Random is more effective than NMRU.

Overall Observations:

- LRU consistently shows high hit ratios across all benchmarks.
- Random replacement policy generally outperforms NMRU.
- The effectiveness of replacement policies varies across benchmarks, indicating that the nature of the workload significantly influences cache performance.

Recommendation:
In scenarios where maximizing hit ratio is crucial, LRU appears to be a strong choice.
Random replacement policy might be a reasonable compromise between simplicity and performance in various scenarios.

Note: These conclusions are based on the provided hit ratios and do not consider other factors such as computational complexity or implementation costs of different replacement policies. The choice of the best policy may depend on specific requirements and constraints in a real-world scenario.


Cache Hit Ratios for Different Benchmarks and Replacement Policies
LRU
NMRU
Random
Hit Ratio (%)
0
20
40
60
80
100
TPCC Oracle
Media Streaming
MapReduce
TPCC DB2
Web Search
Web Frontend
SAT Solver
Data Serving
Benchmarks