Advanced Computer Architecture
Programming Assignment 4

Faculty of Computer Engineering

Abolfazl Bayat
Prof: Dr. Pejman Lotfi-Kamran

Nov 2023
Azar 1402

Contents

Note: If you have already solved Assignment 3 and are familiar with set-assosiative cache simulator, you can skip this sections:

- Cache simulator (implementation)
- Evaluation (implementation)

Introduction to the problem

Overview

In this assignment, you will extend the cache simulator that you developed in PA3 to assess the effective utilization of the cache silicon real estate using traces of CloudSuite and OLTP workloads. Unlike commonly used cache assessment metrics (e.g., hit ratio) that consider the input-output relationship, in this programming assignment, you evaluate a cache based on what happens inside the cache. 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

A block that is kept in a cache, but will not be accessed again until it gets evicted is called a dead block (i.e., every block will become dead after the last access to it, and until it gets evicted from the cache). Extend the cache component that you developed in PA3 to measure what fraction of the cache blocks are dead on average. For this purpose, start the simulation with an empty cache, wait until all cache frames are full (i.e., their valid bit is set), and then counts the number of dead blocks after every access to the cache. At the end of the simulation, report the average number of dead blocks by dividing the sum of the dead blocks that you obtained after every access to the cache over the number of such accesses.

Note : To classify a cache block as dead, you need to know the future accesses to the cache. Remember you can always move forward in the trace file to see the future!

Milestone 1

For every workload, measure the average percentage of dead blocks for a 32 KB, 8-way associative L1d cache with a 64B block size that uses the LRU replacement policy.

Milestone 2 (OPTIONAL but HIGHLY recommended)

Propose an idea to reduce the percentage of dead blocks in a cache. Implement the proposed idea 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 what you observed in this experiment. Please include the percentage of dead blocks for every workload in your report. What did you learn from this experiment? If you have done Milestone 2, please clearly explain the proposed idea, justify why you believe it works, and report the simulation numbers to back up your claim.

cache simulator (implementation)

```
import os
```

- Import the os module to work with operating system functionalities.

```
from collections import OrderedDict
```

- Import the OrderedDict class from the collections module to create an ordered dictionary.

```
import random
```

- Import the random module to generate random numbers.

```
cache = OrderedDict()
```

- Create an ordered dictionary named cache to simulate a cache. It will have sets as keys (s0, s1, ..., s63), and each set initially contains an empty list.

```
for i in range(64):
   cache[f"s{i}"] = []
```

- Initialize each set in the cache with an empty list.

```
block_size = 64
```

- Set the block size for the cache.

```
number_of_set = 64
```

- Set the number of sets in the cache.

```
def replacement(policy):
```

- Define a function replacement that takes a replacement policy as an argument.

```
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)
```

- Inside the function:

- If the policy is "LRU", the least recently used policy is applied by removing the oldest block in the set and appending the new block.
- If the policy is "NMRU" (not most recently used), a random block is removed from the set, and the new block is appended.
- For any other policy, a random block is removed, and the new block is appended.

```
replacement_policies = ["LRU", "NMRU", "RANDOM"]
```

- Create a list of replacement policies.

```
current_directory = os.getcwd()
```

- Get the current working directory.

```
```

```
relative_path = 'traces/'
```

- Set a relative path.

```
main_directory = os.path.join(current_directory, relative_path)
```

- Join the current directory with the relative path to get the main directory path.

```
for subdir, dirs, files in os.walk(main_directory):
```

- Walk through the main directory and iterate over subdirectories, directories, and files.
- Inside the loop:

```
benchmark_path = os.path.join(subdir, benchmark)
```

- Construct the full path to the benchmark file.\

```
bechmark_name = (subdir.split("/"))[-1]
```

- Extract the benchmark name from the subdirectory path.

```
print(f"On {bechmark_name} Benchmark")
```

- Print the benchmark name.
- Inside another loop

```
for policy in replacement_policies:
```

- Iterate over each replacement policy.
- Print the policy being used.

```
with open(benchmark_path) as file:
        hit_count = 0
        miss_count = 0
```

```
            for line in file:
                pc_addr = line.split()
                address = int(pc_addr[1], 16)
                mm_block_number = address // block_size
                set_number = mm_block_number % number_of_set
                if mm_block_number in cache[f"s{set_number}"]:
                    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_count+=1
                    if len(cache[f"s{set_number}"]) < 8:
                        cache[f"s{set_number}"].append(mm_block_number)
                    else:
                        replacement(policy)
```

- Open the benchmark file.
- Initialize hit and miss counters.
- Iterate through each line in the file:
- Extract the address from the line.
- Calculate the block number and set number.
- Check if the block is in the cache set:
- If yes, it's a cache hit, update counters, and adjust the block's position in the set.
- If no, it's a cache miss:
- If the set is not full, add the block to the set.
- If the set is full, apply the replacement policy using the replacement function.

```
    hit_ratio = hit_count / (hit_count + miss_count)
          tmp_hit_ratio = round(hit_ratio, 4)
          hit_ratio = tmp_hit_ratio * 100
          print(f"Hit Ratio = {hit_ratio}%")
          if policy == "RANDOM":
          print("--------------------------------")
```

- Calculate and print the hit ratio for the current policy.
- If the policy is "RANDOM," print a separator line.
- The code simulates cache behavior for different replacement policies on various benchmarks. It reads memory addresses from benchmark files, simulates cache hits and misses, and calculates hit ratios.

Average Dead Blocks (implementation)

```
replacement_policies = ["LRU"]
```

- Instead of considering all replacement policies, only "LRU" is used in this case.

```
cache_sum = 0
```

- Initialize a variable to keep track of the total number of blocks in the cache.

```
hit_count = 0
miss_count = 0
```

- Initialize counters for cache hits and misses.

```
init = 0
```

- A flag to check if the initialization phase is complete.

```
start_here = 0
```

- A variable to store the starting point in the simulation.

```
dead_time = 0
```

- Unused variable, not referenced elsewhere.

```
last_hit_times = {}
```

- Dictionaries to store the starting and ending times for dead blocks.

```
    if benchmark_name == "MapReduce":
                        dead_blocks_trace = [0] * 9332922
                    else:
                        dead_blocks_trace = [0] * 10000000
```

- An array to keep track of the number of times each block is evicted.

```
    if start_here_flag == 0 and len(cache) == 512:
                            start_here = hit_count + miss_count
                            start_here_flag = 1

                        ''' block in cache checking...'''
                        if mm_block_number in cache[f"s{set_number}"]:
                            ''' hit policy + dead policy'''
                            last_hit_times[mm_block_number] = hit_count
+ miss_count
                            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 + dead policy'''
                            last_hit_times[mm_block_number] = hit_count
+ miss_count
                            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:
                                evicted = replacement(policy)
                                evicted_time = hit_count + miss_count
```

```
                                for d in range(last_hit_times[evicted],
evicted_time):
                                    dead_blocks_trace[d] += 1
```

Count the number of dead blocks and calculate the dead block ratio.

Print the separator line only for "LRU" policy.

The code extends the cache simulation to track the starting and ending times of dead blocks, as well as calculate the dead block ratio. Dead blocks are those that are evicted from the cache. The initialization phase ensures that dead blocks are accurately tracked once the cache reaches its maximum size. The dead block ratio is the ratio of the sum of dead blocks to the total number of evicted blocks.

Evaluation (cache simulation)

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:

On TPCC Oracle Benchmark

- with LRU policy Hit Ratio = 93.4%
- with NMRU policy Hit Ratio = 92.61%
- with RANDOM policy Hit Ratio = 92.25999999999999%
- ---

On Media Streaming Benchmark

- with LRU policy Hit Ratio = 94.81%
- with NMRU policy Hit Ratio = 94.33%
- with RANDOM policy Hit Ratio = 94.13%
- ---

On MapReduce Benchmark

- with LRU policy Hit Ratio = 94.82000000000001%
- with NMRU policy Hit Ratio = 94.25%
- with RANDOM policy Hit Ratio = 93.84%
- ---

On TPCC DB2 Benchmark

- with LRU policy Hit Ratio = 93.26%
- with NMRU policy Hit Ratio = 92.7%
- with RANDOM policy Hit Ratio = 92.39%
- ---

On Web Search Benchmark

- with LRU policy Hit Ratio = 98.31%
- with NMRU policy Hit Ratio = 98.14%
- with RANDOM policy Hit Ratio = 98.02%
- ---

On Web Frontend Benchmark

- with LRU policy Hit Ratio = 96.43%
- with NMRU policy Hit Ratio = 96.1%
- with RANDOM policy Hit Ratio = 95.93%
- ---

On SAT Solver Benchmark

- with LRU policy Hit Ratio = 96.93%
- with NMRU policy Hit Ratio = 96.67999999999999%
- with RANDOM policy Hit Ratio = 96.49%
- ---

On Data Serving Benchmark

- with LRU policy Hit Ratio = 94.25%
- with NMRU policy Hit Ratio = 93.92%
- with RANDOM policy Hit Ratio = 93.64%

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.

Evaluation (Average Dead Blocks)

Run the code and evaluate yourself, if the code execution time is too long, use a smaller trace file, the average dead blocks should be around 60-70%.