Computer Organization 2025 Programming Assignment III

Computer Organization 2025 Programming Assignment III

Due Date: 23:59, June 11, 2025

Overview

This assignment aims to enhance program execution performance through effective cache utilization. Modern computing systems leverage caches to reduce memory access latency by storing recently accessed or frequently used data closer to the processor. During the early stages of processor design, it is often the case that designers adjust cache organization and configurations according to application runtime behaviors. Later, when the processor is available, application code is typically optimized to match the hardware characteristics for improved performance. In this assignment, you will practice both of these concepts.

1. **Application-aware cache design** (Hardware perspective)
    - Implement policies for managing limited cache resources
    - Determine which data remains in cache and which gets evicted when cache is full to make room for new data

2. **Cache-architecture aware memory accessing** (Software perspective)
    - Structure code to exploit cache behavior through spatial and temporal locality
    - Organize memory access patterns to maximize cache hit rates

This assignment involves implementing a FIFO cache replacement policy in the Spike simulator and optimizing matrix operations in programs to reduce memory access overhead. In addition, you should also demonstrate your implemented code and explain the design rationale and techniques. This assignment has a total of 110 points: 30 points for code implementation and 80 points for the oral demonstration evaluated by TAs.

Programming Part (30%)

- Implement the **FIFO (First-In-First-Out)** cache replacement policy in the *Spike* simulator
- Optimize the provided programs (matrix transpose and matrix multiply) to reduce memory access overhead

Demo Part (80%)

- Explain the implementation and operation of your FIFO cache replacement policy in Spike
- Present the design philosophy behind your developed code, highlighting the specific techniques and concepts

1. Cache Optimization Fundamentals

Introduce Common Replacement Policies (Hardware Perspective)

As a cache replacement policy can greatly affect the cache hit/miss rate, it has been extensively studied. Consequently, many replacement policies have been proposed, each suitable for certain purposes. The cache hit/miss rate is an important metric for evaluating the efficiency of a cache design. In this assignment, your are required to implement FIFO replacement policy.

The three of the most commonly seen cache replacement policies are listed below.

- First In First Out (FIFO): The cache behaves the same as a FIFO queue, where it evicts the cache blocks in the same order as they were added to the cache without considering any other factors.
- Least Recently Used (LRU): Those the least recently used cache blocks are discarded first.
- Least-Frequently Used (LFU): Those cache blocks that are used least often are replaced first.

For more cache replacement policies (or the above three policies), please refer to the web page for more details.

Introduce Algorithmic Level Method to Reduce Cache Miss (Software Perspective)

The cache miss rate can be influenced by multiple factors. It is not only affected by the replacement policy but also by the way you implement the algorithm. Below, we list common strategies for reducing miss rates that you can apply to your algorithm. The major idea is to improve the spatial or temporal locality.

- Loop Interchage: Change the order you access data to improve the spcial locality on cache access.
- Loop Fusion: Combine the operation on same array at different loop to increase temporal locality.
- Loop Tiling: Breaking down a large loop into smaller blocks or tiles and organizing computations within each tile. Loop tiling enhances the reuse of data stored in the cache.

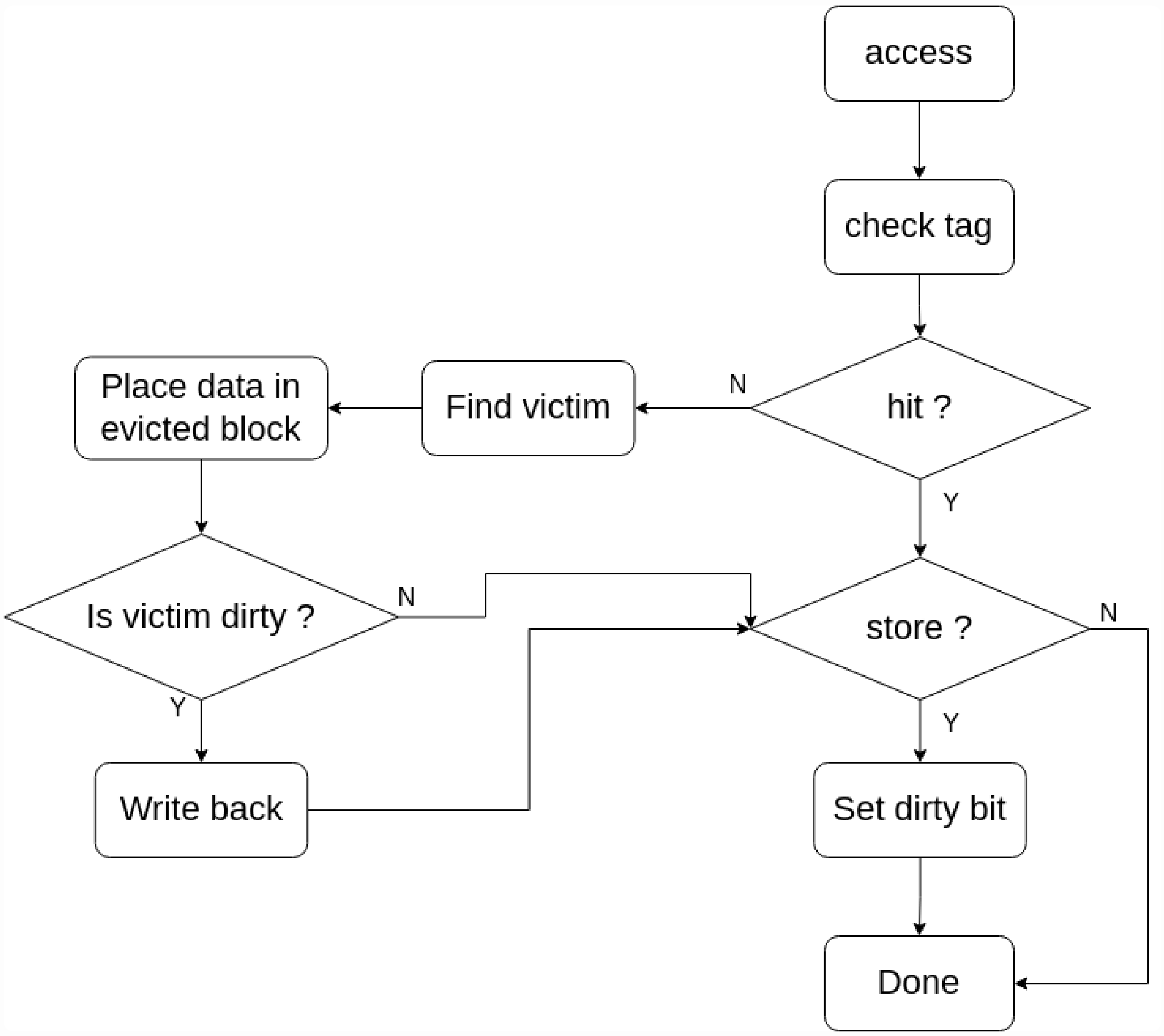
For more details, please refer to the links provided above.

2. Cache Simulation with Spike

During the program emulation by Spike, the emulated instructions (the addresses of the instructions) are fed into the L1 *instruction cache simulator* and the data required by the emulated instructions (e.g., accessed via the memory instructions) are fed to the *L1 data cache simulator*. The cache simulator(s) are represented as the `cache_sim_t` class objects in Spike. The creation of the cache simulators is determined by command line parameters of Spike; for example, the `dc` flag is used to create a L1 data cache simulator, and its parameters are used to specify the attributes of the created data cache, such as *set* and *ways*. Please refer to the following *Section 2 (Assignments) under 'Exercise 1: Implement FIFO Data Cache Replacement Policy' in the Programming Part* for more details.

The `cache_sim_t` class defines the behaviors of the cache simulators created by Spike, where the prototypes are defined in `riscv/cachesim.h` and the implementation is in `riscv/cachesim.cc`. Both data and instruction caches inherit the interfaces defined in the `cache_sim_t` class. There are some important member functions related to manipulate the created cache(s), such as `access`, `check_tag`, and `victimize`. In the following paragraphs, we briefly introduce the high-level concepts of access, and `check_tag`. You should trace the source code to figure out more details of the member functions so as to understand the mechanism and workflow of the caches simulated by the cache_sim_t objects.

The workflow of the data cache simulation in Spike is highlighted as follows. The data cache simulation starts when a memory read/write is emulated by Spike, and the access function is invoked. The prototype of the access member function is `cache_sim_t::access(uint64_t addr, size_t bytes, bool store)`, where the parameter addr contains the address to the to-be-accessed data, bytes represent the size of the to-be-accessed data, and store refers to its a read or store operation. The high level workflow is illustrated in the following image.



As for the `cache_sim_t::check_tag(uint64_t addr)` function, it is used to check the existence of a to-be-accessed data in the simulated data cache (i.e., check tag rectangle in the workflow image). The index to the set for the to-be-accessed data is calculated by the addr argument via the formula: `idx = (addr >> idx_shift) & (sets-1)`. If you have configured a four-set data cache, the idx value is the remainder of the dividing by 4 operation; that is, the value range of idx is from 0 to 3. Next, all of the tag in the corresponding ways are compared with the input argument addr to find the existence of the data with the addr in the data cache. If a match is found, it means a cache it otherwise, it is a cache miss. When a cache miss occurs, the victimize function is invoked to find a victim data cache block to store the new input data. Currently, Spike implements only the random algorithm to randomly select the victim data block.

When cache miss occurrs, the victim block should be selected to store the input data. In particular, Spike leverage the Linear-Feedback Shift Register (LFSR) to implement a pseudo-random cache replacement policy.

Note

Computer simulations do not always replicate real hardware specifications and behaviors. Instead, they focus on some designated problems and use different

> workloads and simulation configurations to evaluate the tradeoffs among different designs. For example, in this assignment, we focus on the L1 data cache designs and try to find out a good cache design that can lead to the lowest data cache miss rate of the given workload. Furthermore, the overhead incurred by the cache misses data is not taken into account, and the implementation overhead (e.g., hardware area, power consumption, computation latency) of different replacement policies is not calculated, too.

Performance Evaluation Methodology

Improvement Ratio is used to quantify the improved performance between different code versions (or cache configurations). The ratio can be calculated via the equation below.

$$\text{Improvement Ratio} = \frac{\text{MemCycle}_{Original}}{\text{MemCycle}_{Improved}}$$

$\text{MemCycle}_{Original}$ indicates the overall cycles of memory access of the original program, while $\text{MemCycle}_{Improved}$ indicates the cycles of memory access of the improved version.

To calculate the memory access overhead, you should use the formula below.

$$\text{MemCycle} = \text{cache}_{hits} \times \text{latency}_{hit} + \text{cache}_{misses} \times \text{penalty}_{miss}$$

The total cycles of memory access overhead are the summation of:

1. the number of cache hit times cache_{hits} multiplied by the latency for a cache hit access latency_{hit}, and
2. the number of cache miss times cache_{misses} multiplied by the cache miss penalty penalty_{miss}.

In this assignment, the data access latencies for a cache hit and a cache miss are configured as follows.

- Cache hit latency (latency_{hit}) = 1 cycle
- Cache miss penalty (penalty_{miss}) = 100 cycles

The following code section lists the performance data of an original version and the improved version. Based on the data below, we can calculate $\text{MemCycle}_{Original}$ and $\text{MemCycle}_{Improved}$, as well as the Improved Ratio.

- $\text{MemCycle}_{Original}$ is $52,269,934$ cycles. $(52,269,934 = ((5,245,118 - 433,720) + (871,214 - 32,478)) \times 1 + (433,720 + 32,478) \times 1$
- $\text{MemCycle}_{Improved}$ is $15,150,540$ cycles.
- Improved Ratio = 3.5 ($\frac{52,269,934}{15,150,540} \approx 3.45$).

An example of performance data for an original version and the improved version. The following performance data is generated by Spike after each simulation run.

```
Original version                           | Improved version
D$ Bytes Read: 24874040                    | D$ Bytes Read: 30530544
D$ Bytes Written: 4006529                  | D$ Bytes Written: 4504069
D$ Read Accesses: 5245118                  | D$ Read Accesses: 6595544
D$ Write Accesses: 871214                  | D$ Write Accesses: 995455
D$ Read Misses: 433720                     | D$ Read Misses: 48792
D$ Write Misses: 32478                     | D$ Write Misses: 27567
D$ Writebacks: 47520                       | D$ Writebacks: 40871
D$ Miss Rate: 7.622%                       | D$ Miss Rate: 1.006%
                                           |
Memory subsystem access overhead = 52269934 | Memory subsystem access overhead = 1

Improved ratio:  3.45003769
```

2. Assignments

This assignment contains two parts and their detailed information is listed as follows.

1. **Programming Part (30%)**
    - **Exercise 1 - Implement FIFO data cache replacement policy (10%)**
    - **Exercise 2 - Enhancement of software programs to reduce memory access overhead (20%)**
        - **Exercise 2-1**: Matrix Transpose (10%)
        - **Exercise 2-2**: 2D Matrix Multiplication (10%)

2. **Demo Part (80%)**
    - Explain how your FIFO cache replacement policy works in Spike
    - Present the design philosophy of your optimized programs, explaining the techniques and concepts applied across all assignment sections

The project files of this assignment will look like this:

```
CO_StudentID_HW3/
├── exercise1
│   ├── cachesim.h
│   ├── cachesim.cc
├── exercise2
│   ├── makefile
│   ├── exercise2_1
│   │   ├── matrix_transpose.c
│   │   ├── matrix_transpose_improved.c
│   │   └── testbench_driver.c
│   ├── exercise2_2
│   │   ├── matrix_multiply.c
│   │   ├── matrix_multiply_improved.c
│   │   ├── testbench_driver.c
│   ├── test_exercise2_1.py
│   └── test_exercise2_2.py
```

Important Notes

- You **must** write the code on your own.
- Write your code only inside the designated sections. Modifications outside these sections are not permitted unless explicitly allowed.
- **Hidden test cases** will be used to evaluate your developed code. Please ensure your programs run correctly under various inputs.
- The provided code contains only essential elements for explaining the assignment. Download the complete project files from NCKU Moodle for development.
- Remember to check the course announcements for the latest updates and reminders.

Programming Part (30%)

Exercise 1. Implement FIFO Data Cache Replacement Policy (10%)

You will implement the FIFO data cache replacement policy in the Spike simulator. This requires the modification of the cache simulation code in the Spike simulator and rebuilding the simulator.

Implementation Requirements

You should use the Spike source code from the `riscv-isa-sim` directory that you installed in HW0. Specifically, you need to modify the following files for cache simulation.

- **`riscv/cachesim.h`**: Add required data fields to track essential information for FIFO cache simulation, such as tracking the order in which cache blocks were added.
    - The relevant code of `class cache_sim_t` is listed as follows.

```
class cache_sim_t
{
 public:
  cache_sim_t(size_t sets, size_t ways, size_t linesz, const char* name);
  cache_sim_t(const cache_sim_t& rhs);
  virtual ~cache_sim_t();

  void access(uint64_t addr, size_t bytes, bool store);
  void clean_invalidate(uint64_t addr, size_t bytes, bool clean, bool inval);
  void print_stats();
  void set_miss_handler(cache_sim_t* mh) { miss_handler = mh; }
  void set_log(bool _log) { log = _log; }

  static cache_sim_t* construct(const char* config, const char* name);

 protected:
  static const uint64_t VALID = 1ULL << 63;
  static const uint64_t DIRTY = 1ULL << 62;

  virtual uint64_t* check_tag(uint64_t addr);
  virtual uint64_t victimize(uint64_t addr);

  lfsr_t lfsr;
  cache_sim_t* miss_handler;
  size_t sets;
  size_t ways;
  size_t linesz;
  size_t idx_shift;

  uint64_t* tags;

  uint64_t read_accesses;
  uint64_t read_misses;
  uint64_t bytes_read;
  uint64_t write_accesses;
  uint64_t write_misses;
  uint64_t bytes_written;
  uint64_t writebacks;

  std::string name;
  bool log;

  void init();
};
```

- **`riscv/cachesim.cc`**: Implement the FIFO replacement mechanism by modifying appropriate member functions of the `cache_sim_t` class. Your implementation should follow the cache simulation workflow described above.
  - The current implementation of `victimize` method (which you need to modify) is shown below.

    ```
    uint64_t cache_sim_t::victimize(uint64_t addr)
    {
        size_t idx = (addr >> idx_shift) & (sets-1);
        size_t way = lfsr.next() % ways;
        uint64_t victim = tags[idx*ways + way];
        tags[idx*ways + way] = (addr >> idx_shift) | VALID;
        return victim;
    }
    ```

  - Additionally, you need to modify the fully-associative cache implementation.

    ```
      uint64_t fa_cache_sim_t::victimize(uint64_t addr)
      {
          uint64_t old_tag = 0;
          if (tags.size() == ways)
          {
              auto it = tags.begin();
              std::advance(it, lfsr.next() % ways);
              old_tag = it->second;
              tags.erase(it);
          }
          tags[addr >> idx_shift] = (addr >> idx_shift) | VALID;
          return old_tag;
      }
    ```

You are allowed to modify the two files: `riscv/cachesim.cc` and `riscv/cachesim.h`. **Please copy the two modified files to the assignment's `exercise1` directory**.

Notes

- Since Spike is written in C++, you can leverage the Standard Template Library (STL) data structures to facilitate your implementation. For example, the `queue` data structure is particularly suitable for implementing the FIFO replacement policy.
- Ensure your Spike simulator has been rebuilt with your FIFO cache implementation before testing.
- If you **cannot** provide a valid implementation for the FIFO data cache replacement policy, you can upload the random policy offered by the Spike simulator. In this case, you can still work on the enhancement of software programs. Of course, you will lose **10** points from this part for not being able to provide a FIFO data cache replacement policy.

Exercise 2. Enhancing Software Programs to Reduce Memory Access Overhead (20%)

You will optimize software programs to reduce memory access overhead based on the given cache configuration. Your optimized versions will be evaluated using the FIFO cache replacement policy you implemented in Exercise 1.

Cache Configuration and Performance Characteristics

You must use the following L1 data cache configuration for all experiments.

- **4 ways**
- **8 sets**
- **32-byte cacheline**
- Cache hit latency = 1 cycle
- Cache miss penalty = 100 cycles

You should use the following command to run a cache simulation using the above setting.

```
$ spike --isa=RV64GC --dc=8:4:32 $RISCV/riscv64-unknown-elf/bin/pk <your_program>
```

Exercise 2-1: Matrix Transpose Optimization (10%)

Background

Matrix transpose involves converting rows into columns and vice versa. For an $m \times n$ matrix $A = [a_{ij}]$, its transpose version $A^T = [a_{ji}]$ is an $n \times m$ matrix.

Requirements of this Exercise

1. Optimize the matrix transpose implementation to reduce memory access cycles.
2. Implement your improved algorithm in `exercise2/exercise2_1/matrix_transpose_improved.c`.

Original Code.

```
void matrix_transpose(int n, int *dst, int *src) {
    for (int x = 0; x < n; x++) {
        for (int y = 0; y < n; y++) {
            dst[y + x * n] = src[x + y * n];
        }
    }
}
```

Input

- Matrix size: $m \times n$, where m and n set to 1,000 (1,000 × 1,000).
- The matries maintain integer type `int` data, and the values of the matrix elements are ranging from 0 to $1,023$.

Output

The example output contains the performance data for both of the original and improved programs.

```
=============== Exercise 2-1 ===============
Original version
D$ Bytes Read:             102247719
D$ Bytes Written:          49349943
D$ Read Accesses:          21301733
D$ Write Accesses:         8675759
D$ Read Misses:            1137185
D$ Write Misses:           702760
D$ Writebacks:             756764
D$ Miss Rate:              6.138%

Memory subsystem access overhead =  212132047 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:             121078975
D$ Bytes Written:          50458427
D$ Read Accesses:          26012881
D$ Write Accesses:         8953944
D$ Read Misses:            253802
D$ Write Misses:           684730
D$ Writebacks:             712532
D$ Miss Rate:              2.684%

Memory subsystem access overhead =  127881493 (cpu cycle)
-----------------------------------
Improved ratio:  1.658817409959391
Output Correctness:  Pass
-----------------------------------
```

Scoring Criteria

- Your obtained scores of this exercise is determined by the improved ratio achieved by your implementation.
    - `Improvement ratio > 1.6`: 10 points.

    - `Improvement ratio > 1`: 5 points.
- If the result of the matrix transpose is incorrect, you will not get the scores.
- **NOTE:** When judging the exercise, we will replace the input matrix size with hidden test cases to verify the correctness of your developed code.

Exercise 2-2: Matrix Multiplication Optimization (10%)

Background

Matrix multiplication combines two matrices to produce a third matrix. For matrices A and B, their product matrix C is mathematically expressed as $C_{ij} = \sum_k A_{ik} \times B_{kj}$.

Requirements of this Exercise

1. Optimize the matrix multiplication implementation to reduce memory access cycles
2. Implement your improved algorithm in `exercise2/exercise2_2/matrix_multiply_improved.c`

Original Code.

```
void matrix_multiply(int *A, int *B, int *Output, int i, int k, int j) {
    for (int x = 0; x < i; x++) {
        for (int y = 0; y < j; y++) {
            int sum = 0;
            for (int z = 0; z < k; z++) {
                sum += A[x * k + z] * B[z * j + y];
            }
            Output[x * j + y] = sum;
        }
    }
}
```

Input

- Both of the two input matrices share the same setting as follows.
- Matrix size: $m \times n$, where m and n set to 100 (100 $\times$ 100).
- The matries maintain integer type `int` data, and the values of the matrix elements are ranging from 0 to $1,023$.

Output

The example output contains the performance data for both of the original and improved programs.

```
=============== Exercise 2-2 ===============
Original version
D$ Bytes Read:            66304079
D$ Bytes Written:         9311467
D$ Read Accesses:         14416460
D$ Write Accesses:        2216132
D$ Read Misses:           1220580
D$ Write Misses:          28107
```

```
D$ Writebacks:              66502
D$ Miss Rate:               7.507%

Memory subsystem access overhead =  140252605 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:              104168227
D$ Bytes Written:           12577667
D$ Read Accesses:           23369163
D$ Write Accesses:          3031046
D$ Read Misses:             111400
D$ Write Misses:            26246
D$ Writebacks:              67375
D$ Miss Rate:               0.521%

Memory subsystem access overhead =  40027163 (cpu cycle)
-----------------------------------
Improved ratio:  3.5039356898714007
Output Correctness:  Pass
-----------------------------------
```

Scoring Criteria

- Your obtained scores of this exercise is determined by the improved ratio by your implementation.
  - `Improvement ratio > 3`: 10 points.
  - `Improvement ratio > 1.5`: 5 points.
- If the result of the matrix multiplication is incorrect, you will not get the scores below.
- **NOTE:** When judging the exercise, we will replace the input matrices with hidden test cases to verify the correctness of your developed code.

Demo Part (80%)

Your job is to register a demo session, which will be announced later, and elaborate your design philosophy of your implementation. **Your score is based on the answers you provide to the TA**.

You have to provide your following answers during the demo session.

1. Describe the workflow and mechanism in Spike, related to cache simulation. (20%)
2. Describe the concept behind your modified matrix transpose algorithm. (20%)
3. Describe the concept behind your modified matrix multiplication algorithm. (20%)
4. Describe the concept behind your design philosophy of previous **Assignment I** and **Assignment II**. (20%)

3. About Hidden Test Cases

There are two hidden test cases for each exercise. For each exercise, the public test case account for 60% of the total score, and **the hidden test cases account for 40%**.

For example, if your code passes Exercise 2-1 with the public test case, you will earn 4 points. If it also passes the two hidden test cases, you will earn the full 10 points.

> **NOTE**
>
> The hidden test cases follow the same value range and the size of matrix as specified in the input description of each exercise.

4. Test Your Assignment

The local-judge system is used to check the results of your exercise 2's developed code. You can run your developed programs and validate their results via the make commands below under `/exercise2` directory. The following example commands can do individual tests for each exercise with public test cases.

- Test your code of whole exercise 2 implementation

```
make check
```

- Test your code of `exercise2_1` with public test cases

```
make checkex2-1
```

- Test your code of `exercise2_2` with public test cases

```
make checkex2-2
```

Example of Output

- **Pass**

```
=============== Exercise 2-1 ===============
Original version
D$ Bytes Read:              102247719
D$ Bytes Written:           49349943
D$ Read Accesses:           21301733
D$ Write Accesses:          8675759
D$ Read Misses:             1137185
D$ Write Misses:            702760
D$ Writebacks:              756764
D$ Miss Rate:               6.138%

Memory subsystem access overhead =  212132047 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:              121078975
D$ Bytes Written:           50458427
D$ Read Accesses:           26012881
D$ Write Accesses:          8953944
D$ Read Misses:             253802
```

```
D$ Write Misses:           684730
D$ Writebacks:             712532
D$ Miss Rate:              2.684%

Memory subsystem access overhead =  127881493 (cpu cycle)
-----------------------------------
Improved ratio:  1.658817409959391
Output Correctness:  Pass
-----------------------------------
=============== Exercise 2-2 ===============
Original version
D$ Bytes Read:             66304079
D$ Bytes Written:          9311467
D$ Read Accesses:          14416460
D$ Write Accesses:         2216132
D$ Read Misses:            1220580
D$ Write Misses:           28107
D$ Writebacks:             66502
D$ Miss Rate:              7.507%

Memory subsystem access overhead =  140252605 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:             104168227
D$ Bytes Written:          12577667
D$ Read Accesses:          23369163
D$ Write Accesses:         3031046
D$ Read Misses:            111400
D$ Write Misses:           26246
D$ Writebacks:             67375
D$ Miss Rate:              0.521%

Memory subsystem access overhead =  40027163 (cpu cycle)
-----------------------------------
Improved ratio:  3.5039356898714007
Output Correctness:  Pass
-----------------------------------
```

- Error

```
=============== Exercise 2-1 ===============
Original version
D$ Bytes Read:             102247719
D$ Bytes Written:          49349943
D$ Read Accesses:          21301733
D$ Write Accesses:         8675759
D$ Read Misses:            1137185
D$ Write Misses:           702760
D$ Writebacks:             756764
D$ Miss Rate:              6.138%

Memory subsystem access overhead =  212132047 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:             46214339
D$ Bytes Written:          41337375
D$ Read Accesses:          9295055
D$ Write Accesses:         6673296
D$ Read Misses:            64515
D$ Write Misses:           546632
```

```
D$ Writebacks:              564360
D$ Miss Rate:               3.827%

Memory subsystem access overhead =  76471904 (cpu cycle)
-----------------------------------
Improved ratio:  2.773986731126768
Output Correctness:  Fail
-----------------------------------
=============== Exercise 2-2 ===============
Original version
D$ Bytes Read:              66304079
D$ Bytes Written:           9311467
D$ Read Accesses:           14416460
D$ Write Accesses:          2216132
D$ Read Misses:             1220580
D$ Write Misses:            28107
D$ Writebacks:              66502
D$ Miss Rate:               7.507%

Memory subsystem access overhead =  140252605 (cpu cycle)
-----------------------------------
Improved version
D$ Bytes Read:              1928931
D$ Bytes Written:           1174355
D$ Read Accesses:           324338
D$ Write Accesses:          179113
D$ Read Misses:             5323
D$ Write Misses:            18915
D$ Writebacks:              20841
D$ Miss Rate:               4.814%

Memory subsystem access overhead =  2903013 (cpu cycle)
-----------------------------------
Improved ratio:  48.312771937294116
Output Correctness:  Fail
-----------------------------------
```

Debugging Failed Tests

If you receive a "Fail" on the Output Correctness check, you can use the following commands to find out detailed information.

- `make diffex2-1` for Exercise 1
- `make diffex2-2` for Exercise 2

These commands will compare your output (`*/stu.output`) with the expected output (`*/ans.output`).

5. Submission of Your Assignment

Your developed codes should be put into the folder: `StudentID_HW3` . Please follow the instructions below to submit your programming assignment.

1. Compress your source code within the folder into a zip file.
2. Submit your homework with NCKU Moodle.

3. The zipped file and its internal directory organization of your developed code should be similar to the example below.

> **NOTE**
>
> Replace all CO_StudentID with your student ID number, e.g., F12345678_HW3.zip

```
F12345678_HW3.zip
└── F12345678_HW3
    ├── README.md
    ├── exercise1
    │   ├── cachesim.cc
    │   └── cachesim.h
    └── exercise2
        ├── exercise2_1
        │   ├── matrix_transpose.c
        │   ├── matrix_transpose_improved.c
        │   └── testbench_driver.c
        ├── exercise2_2
        │   ├── matrix_multiply.c
        │   ├── matrix_multiply_improved.c
        │   └── testbench_driver.c
        ├── makefile
        ├── test_exercise2_1.py
        └── test_exercise2_2.py
```

- Do not submit any files that are not listed above.
- In addition to your code, you must also submit a file named README.md. This document should record your development process to prove that the submitted code is your own work. README.md is only accepted in Markdown or plain text format. You can use HackMD to edit your README.md file. README.md can be written in Chinese or English
- Please maintain the filename as README.md only. Do not modify it by adding any additional information such as your student ID.
- A plagiarism checking process will be performed on your submitted code. A high similarity score will result in a score of zero.

> **Note**
>
> **You can include draft answers to the four questions of the demo part in your README.**
>
> **During the demo session, you are allowed to reference your README file to help you answer TA's questions.**

!!! Incorrect format (either the file structure or file name) will lose 10 points. !!!
!!! A 30% penalty will be applied for late submissions within seven days (from 00:00, June 12 to 23:59, June 18, 2025) after the deadline. !!!
!!! Do not modify the `Makefile` , `testbench_driver.c` , `test_exercise_1.py` and

`test_exercise_2.py`**, as this may cause the judge program to fail, resulting in a score of zero. !!!**

6. References

- **Cache Replacement Policies**
- **Loop Interchange**:
- **Loop Fusion**
- **Loop Tiling**
- **Linear-Feedback Shift Register (LFSR)**