# Homework #4: Cache

Due: May. 18, 11:59 PM
Responsible TA: Jongyoung Kang (jongyoung_kang@unist.ac.kr)

- **Several updated notes:**
  - 5/12 Update: There is a typo in Questions 2(c) and 2(d): `eval-transpose` should be `eval-simulator`.

**Our position on AI use:**

You may use generative AI tools only in the limited role of **a search engine substitute (e.g., instead of Google)**. This means you may ask AI tools for explanations of concepts or for help finding reference material, but **you may not request or use code of any form.**

AI tools are a part of today's world and will be an even larger part of the one you'll enter after you graduate. Our goal is to help you build skills and understanding that surpass what AI can offer, so that you will not be replaceable by AI. The restrictions that are in place for this course are there because we believe that an over-reliance on AI would prevent you from fully achieving the learning objectives of the course and developing the foundational knowledge you will need to be successful in your future coursework and career. Ultimately, the value you gain from this course will be directly related to the effort you put in. Generally, the more you rely on AI, the less you'll learn. Use AI thoughtfully, if and when it is permitted, but never as a substitute for your own learning.

⋆ We will collect code generated by AI tools, including Claude, ChatGPT, Gemini, and others, as well as code publicly available on GitHub, blogs, and similar sources, and use them for plagiarism detection. If you rely on generated or copied code, it will be detected. If plagiarism is detected in any form, you will receive an F immediately.

# 1 Homework Description

## 1.1 Overview

The HW4 consists of three parts. In the first part (§1.4), you will write a small C program (about 200–300 lines) that simulates the behavior of a cache memory. In the second part (§1.5), you will optimize a matrix transpose function, with the goal of minimizing the number of cache misses. Finally, you must answer all the given questions (§2).

## 1.2 Downloading the Assignment

Please download the handout named `hw4.zip` [1] and copy it to the home directory of your assigned server, where you plan to work. Then run the following command:

```
$ unzip hw4.zip
```

This will create a directory called `hw4` that contains a number of files. You will be modifying two files: `simulator.c` and `transpose.c`. To compile these files, type:

```
$ make clean
$ make
```

## 1.3 Reference Trace Files

The `traces` subdirectory of the handout directory contains a collection of reference trace files that we will use to evaluate the correctness of the cache simulator you write in Part A. The trace files are generated by a Linux program called `valgrind`. For example, typing

```
$ valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

on the command line runs the executable program `ls -l`, captures a trace of each of its memory accesses in the order they occur, and prints them on `stdout`.

Valgrind memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is:

```
[space]operation address,size
```

The `operation` field denotes the type of memory access: `I` denotes an instruction load, `L` a data load, `S` a data store, and `M` a data modify (i.e., a data load followed by a data store). There is never a space before each `I`. There is always a `[space]` before each `M`, `L`, and `S`. The `address` field specifies a 64-bit hexadecimal memory address. The `size` field specifies the number of bytes accessed by the operation.

## 1.4 Part A: Writing a Cache Simulator

In Part A, you will write a cache simulator in `simulator.c` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

We have provided you with the binary executable of a reference cache simulator, called `simulator-ref`, that simulates the behavior of a cache with **arbitrary size** and `associativity` on a `valgrind` trace file. It supports both LRU (Least-Recently Used) and LFU (Least-Frequently Used) replacement policies.

The reference simulator takes the following command-line arguments:

```
Usage: ./simulator-ref [-hv] [-R lru|lfu] -S <num> -N <num> -B <num> -t <file>
```

- `-h`: Optional help flag that prints usage information.
- `-v`: Optional verbose flag that displays trace information.
- `-R <lru|lfu>`: Replacement policy (lru or lfu).
- `-S <num>`: Number of set index bits ($2^S$ is the number of sets).
- `-N <num>`: Number of blocks per set (associativity or number of ways).
- `-B <num>`: Number of block offset bits (the block size is $2^B$ bytes).
- `-t <tracefile>`: Name of the `valgrind` trace to replay.

For example:

```
$ ./simulator-ref -R lru -S 4 -N 1 -B 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
$ ./simulator-ref -v -R lru -S 4 -N 1 -B 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Your job for Part A is to fill in `simulator.c` so that it takes the same command-line arguments and produces the identical output as `simulator-ref` for both replacement policies. Note that this file is almost completely empty. You will need to write it from scratch.

**Programming Rules for Part A**

- Your `simulator.c` file must compile without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary `-S`, `-N`, and `-B` values. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function. Type `man malloc` for information about this function.
- For this homework, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with `I`). Recall that valgrind always puts `I` in the first column (with no preceding `[space]`), and `M`, `L`, and `S` in the second column (with a preceding `[space]`). This may help you parse the trace.
- To receive credit for Part A, you must call the function `printSummary` (declared in `hw4_lib.c`), with the total number of hits, misses, and evictions, at the end of your main function:

  ```
  printSummary(hit_count, miss_count, eviction_count);
  ```

- For this homework, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the valgrind traces.
- Your simulator must support both the LRU and LFU replacement policies, selected via the `-R` option. When `-R` is not specified, your simulator should default to LRU.

## 1.5 Part B: Optimizing Matrix Transpose

In Part B, you will write a transpose function in `transpose.c` that causes as few cache misses as possible. Let $A$ denote a matrix, and $A_{ij}$ denote the component on the $i$-th row and $j$-th column. The transpose of $A$, denoted $A^T$, is a matrix such that $A_{ij} = A^T_{ji}$.

To help you get started, we have given you an example transpose function in `transpose.c` that computes the transpose of an $N \times M$ matrix $A$ and stores the results in an $M \times N$ matrix $B$:

```
char transpose_desc[] = "Simple row-wise scan transpose";
void transpose(int M, int N, int A[N][M], int B[M][N])
```

The example transpose function is correct, but it is inefficient because the access pattern results in relatively many cache misses. Your job in Part B is to write a similar function, called `transpose_submit`, that minimizes the number of cache misses across different sized matrices:

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

Do *not* change the description string (`"Transpose submission"`) for your `transpose_submit` function. The autograder searches for this string to determine which transpose function to evaluate for credit.

**Programming Rules for Part B**

- Your code in `transpose.c` must compile without warnings to receive credit.
- You are allowed to define at most 12 local variables of type `int` per transpose function[1].
- You are not allowed to side-step the previous rule by using any variables of type `long` or by using any bit tricks to store more than one value in a single variable.
- Your transpose function may not use recursion.
- If you choose to add functions, you may not have more than 12 local variables on the stack at a time between your added functions and your top-level transpose function. For example, if your transpose declares 8 variables, and then you call a function which uses 4 variables, which calls another function which uses 2, you will have 14 variables on the stack, and you will be in violation of the rule.
- Your transpose function may not modify array *A*. You may, however, do whatever you want with the contents of array *B*.
- You are **NOT** allowed to define any arrays in your code or to use any variant of `malloc`.

## 1.6 Evaluation

The full score for this homework is **200 points**:

- §1.6.1 Part A (Simulator Correctness): 66 Points (33 for LRU + 33 for LFU)
- §1.6.2 Part B (Transpose Performance): 61 Points
- §1.6.3 Coding Style: 7 Points
- §2 Document (`report.pdf`): 66 points

### 1.6.1 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters, traces, and replacement policies. There are nine test cases per policy (LRU and LFU), giving a total of 66 points. Each test case (except the last two) is worth 3 points; the last two cases are worth 6 points each. You can use `eval-simulator` to check your results:

```
$ make
$ ./eval-simulator
```

When all test cases pass, the output will look like this:

```
[ lru ]  |           |     Your simulator      |   Reference simulator   |
Points   | (S,N,B)   | Hits    Misses  Evicts  | Hits    Misses  Evicts  | Trace
---------+-----------+-------------------------+-------------------------+--------------------
3/3      | (1,1,1)   | 9       8       6       | 9       8       6       | traces/yi2.trace
3/3      | (4,2,4)   | 4       5       2       | 4       5       2       | traces/yi.trace
3/3      | (2,1,4)   | 2       3       1       | 2       3       1       | traces/dave.trace
3/3      | (2,1,3)   | 167     71      67      | 167     71      67      | traces/transpose.trace
3/3      | (2,2,3)   | 201     37      29      | 201     37      29      | traces/transpose.trace
3/3      | (2,4,3)   | 212     26      10      | 212     26      10      | traces/transpose.trace
3/3      | (5,1,5)   | 231     7       0       | 231     7       0       | traces/transpose.trace
6/6      | (5,1,5)   | 265189  21775   21743   | 265189  21775   21743   | traces/long.trace
6/6      | (5,4,5)   | 279354  7610    7482    | 279354  7610    7482    | traces/long.trace
---------+-----------+-------------------------+-------------------------+--------------------
33/33    |
```

[1]The reason for this restriction is that our testing code is not able to count references to the stack. We want you to limit your references to the stack and focus on the access patterns of the source and destination arrays.

```
[ lfu ]  |           |     Your simulator      |   Reference simulator   |
Points   | (S,N,B)   | Hits    Misses  Evicts  | Hits    Misses  Evicts  | Trace
---------+-----------+-------------------------+-------------------------+--------------------
3/3      | (1,1,1)   | 9       8       6       | 9       8       6       | traces/yi2.trace
3/3      | (4,2,4)   | 5       4       1       | 5       4       1       | traces/yi.trace
3/3      | (2,1,4)   | 2       3       1       | 2       3       1       | traces/dave.trace
3/3      | (2,1,3)   | 167     71      67      | 167     71      67      | traces/transpose.trace
3/3      | (2,2,3)   | 202     36      28      | 202     36      28      | traces/transpose.trace
3/3      | (2,4,3)   | 209     29      13      | 209     29      13      | traces/transpose.trace
3/3      | (5,1,5)   | 231     7       0       | 231     7       0       | traces/transpose.trace
6/6      | (5,1,5)   | 265189  21775   21743   | 265189  21775   21743   | traces/long.trace
6/6      | (5,4,5)   | 268108  18856   18728   | 268108  18856   18728   | traces/long.trace
---------+-----------+-------------------------+-------------------------+--------------------
33/33    |
```

For each test case, outputting the correct number of cache hits, misses, and evictions will give you full credit for that test case. Each of the three reported values (hits, misses, evictions) is worth 1/3 of the credit for that test case. You can use the reference simulator `simulator-ref` with the `-v` option for a detailed record of each hit and miss during debugging.

#### 1.6.2 Evaluation for Part B

For Part B, we will evaluate the correctness and performance of your `transpose_submit` function on three different-sized matrices:

- $32 \times 32$ ($M = 32$, $N = 32$)
- $64 \times 64$ ($M = 64$, $N = 64$)
- $61 \times 67$ ($M = 61$, $N = 67$)

For each matrix size, the performance of your `transpose_submit` function is evaluated by using `valgrind` to extract the address trace for your function, and then using `simulator-ref` to replay this trace on a cache with parameters ($S = 5$, $N = 1$, $B = 5$).

Your performance score for each matrix size scales linearly with the number of misses $m$, up to some threshold:

- $32 \times 32$: 19 points if $m < 300$, 0 points if $m > 600$
- $64 \times 64$: 19 points if $m <$ 1,300, 0 points if $m >$ 2,000
- $61 \times 67$: 23 points if $m <$ 2,000, 0 points if $m >$ 3,000

Your code must be correct to receive any performance points for a particular size. Your code only needs to be correct for these three cases and you can optimize it specifically for these three cases. In particular, it is perfectly OK for your function to explicitly check for the input sizes and implement separate code optimized for each case.

#### 1.6.3 Evaluation for Coding Style

We have reserved 7 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Also, your comments should be informative, but not be excessive. We will inspect your code for illegal arrays and excessive local variables.

### 1.7 Working on the Homework

#### 1.7.1 Working on Part A

We have provided you with an autograding program, called `eval-simulator`, that tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
$ make clean
$ make
$ ./eval-simulator
```

Here are some hints and suggestions for working on Part A:

- Do your initial debugging on the small traces, such as `traces/dave.trace`.
- The reference simulator `simulator-ref` takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. You are not required to implement this feature in your `simulator.c`, but we strongly recommend that you do so. It will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.
- We recommend that you use the `getopt` function to parse your command-line arguments. You'll need the following header files:

  ```
  #include <getopt.h>
  #include <stdlib.h>
  #include <unistd.h>
  ```

  See `man 3 getopt` for details.
- Each data load (`L`) or store (`S`) operation can cause at most one cache miss. The data modify operation (`M`) is treated as a load followed by a store to the same address. Thus, an `M` operation can result in two cache hits, or a miss and a hit plus a possible eviction.
- For the LFU policy, evict the cache line with the smallest access frequency. In case of a tie in frequency, use LRU as a tiebreaker (i.e., among the least-frequently used lines, evict the one that was least recently used).

### 1.7.2 Working on Part B

We have provided you with an autograding program, called `eval-transpose`, that tests the correctness and performance of each of the transpose functions that you have registered with the autograder.

You can register up to 100 versions of the transpose function in your `transpose.c` file. Each transpose version has the following form:

```
/* Header comment */
char transpose_simple_desc[] = "A simple transpose";
void transpose_simple(int M, int N, int A[N][M], int B[M][N])
{
    /* your transpose code here */
}
```

Register a particular transpose function with the autograder by making a call of the form:

```
registerTransFunction(transpose_simple, transpose_simple_desc);
```

in the `registerFunctions` routine in `transpose.c`. At runtime, the autograder will evaluate each registered transpose function and print the results. Of course, one of the registered functions must be the `transpose_submit` function that you are submitting for credit:

```
registerTransFunction(transpose_submit, transpose_submit_desc);
```

See the default `transpose.c` function for an example of how this works.

The autograder takes the matrix size as input. It uses valgrind to generate a trace of each registered transpose function. It then evaluates each trace by running the reference simulator on a cache with parameters ($S = 5, N = 1, B = 5$).

For example, to test your registered transpose functions on a $32 \times 32$ matrix, rebuild `eval-transpose`, and then run it with the appropriate values for $M$ and $N$ :

```
$ make clean
$ make
$ ./eval-transpose -M 32 -N 32
```

The `eval-transpose` program tests each of the registered functions, displays the results for each, and extracts the results for the official submission.

Here are some hints and suggestions for working on Part B:

- The `eval-transpose` program saves the trace for function *i* in file `trace.fi`[2]. These trace files are invaluable debugging tools that can help you understand exactly where the hits and misses for each transpose function are coming from. To debug a particular function, simply run its trace through the reference simulator with the verbose option:

  ```
  $ ./simulator-ref -v -R lru -S 5 -N 1 -B 5 -t trace.f0
  S 10d0a0,1 miss
  L 18d0e0,8 miss
  L 18d0c4,4 miss
  L 18d0c0,4 hit
  L 10d0c0,4 miss eviction
  L 10d0c4,4 hit
  ...
  ```

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem. Think about the potential for conflict misses in your code, especially along the diagonal. Try to think of access patterns that will decrease the number of these conflict misses.

### 1.7.3 Putting it all Together

We have provided you with a driver program, called `auto-grade.py`, that performs a complete evaluation of your simulator and transpose code. This is the same program your instructor uses to evaluate your submissions. The driver uses `eval-simulator` to evaluate your simulator and `eval-transpose` to evaluate your submitted transpose function on the three matrix sizes. Then it prints a summary of your results and the points you have earned.

To run the driver, type:

```
$ ./auto-grade.py
```

When all parts are complete, the output will resemble:

```
Part A: Testing cache simulator
Running ./eval-simulator
...
Part B: Testing transpose function
Running ./eval-transpose -M 32 -N 32
Running ./eval-transpose -M 64 -N 64
Running ./eval-transpose -M 61 -N 67

HW4 grading summary:
                        Points   Max pts      Misses
Simualtor correctness     66.0        66
Transpose perf 32x32      19.0        19          65
Transpose perf 64x64      19.0        19         129
Transpose perf 61x67      23.0        23         123
          Total points   127.0       127
```

[2]Because valgrind introduces many stack accesses that have nothing to do with your code, we have filtered out all stack accesses from the trace. This is why we have banned local arrays and placed limits on the number of local variables.

# 2 Questions (66 points)

(a) Describe in detail how you implemented `simulator.c`. (15 poitns)

(b) Describe in detail how you implemented `transpose.c`. (15 poitns)

(c) Based on your observations from ~~`eval-transpose`~~`eval-simulator`, when do LRU and LFU produce the same results? Why do they produce the same results in those cases? (5 points)

(d) Based on your observations from ~~`eval-transpose`~~`eval-simulator`, when do LRU and LFU produce different results? Which replacement policy performs better? Explain why the policy you selected can be considered better. (5 points)

(e) Consider the following `S`, `N`, and `B` configurations.

| S | N | B (Block Size) | Miss Rate |
|---|---|---|---|
| 7 | 1 | 3 (8 bytes) | |
| 6 | 1 | 4 (16 bytes) | |
| 5 | 1 | 5 (32 bytes) | |
| 4 | 1 | 6 (64 bytes) | |
| 3 | 1 | 7 (128 bytes) | |

(1) Report the tag size for each configuration. Are all tag sizes the same? Explain why or why not. (4 points)

(2) Are all cache sizes the same? Explain why or why not. (4 points)

(3) Fill in the miss rate for each configuration. Then, draw a line graph where the x-axis is the block size and the y-axis is the miss rate. (4 points)

(4) Based on the graph, what trend do you observe as the block size increases? Is increasing the block size always beneficial? Explain all lessons you can draw from this experiment. (5 points)

(f) Consider the following `S`, `N`, and `B` configurations.

| S | N | B | Miss Rate |
|---|---|---|---|
| 6 | 1 | 3 | |
| 6 | 3 | 3 | |
| 6 | 6 | 3 | |
| 6 | 9 | 3 | |
| 6 | 12 | 3 | |

(1) Fill in the miss rate for each configuration. Then, draw a line graph where the x-axis is the set associativity and the y-axis is the miss rate. (4 points)

(2) Based on the graph, what trend do you observe as the set associativity increases? Is increasing the set associativity always beneficial? Explain all lessons you can draw from this experiment. (5 points)

# 3 Environment

To complete this homework, you must access the server at `10.20.18.55` via `ssh` using the account information provided to you. Note that this server can only be accessed from the UNIST internal network, so make sure to use a VPN if you are connecting from outside.

Please also note that grading will be conducted in the provided server environment. In other words, your code must compile and run correctly on our server. If your code does not compile or does not work correctly in our server environment, you will lose points. No exceptions will be made.

# 4 Submission Guidelines

- You should submit a single zip archive through Blackboard, named in the format `Your_ID-hw4.zip`. For example, if your student ID is 20231234, then your submission file should be named `20231234-hw4.zip`. The zip archive must include the following files:
  - `simulator.c` and `transpose.c`: your completed versions based on the provided skeleton code [1].
  - `report.pdf`: your report must answer every question (§2) in this handout. The report may be written in either English or Korean.

  If the required naming convention is not followed exactly, the submission will receive a score of zero. No exceptions will be made.
- **Late submission policy.** Late submission will be assessed a penalty of 10% per day (We will only accept late submissions of up to 3 days).

# Note

- **Be careful about plagiarism!** We will conduct strict cross-plagiarism detection for evaluation, including a series of answers generated by ChatGPT, code found online, and your submissions. Therefore, we do not recommend consulting ChatGPT or online solutions. Last semester, we identified several cases of plagiarism using an automated tool. Do not share your code or intermediate results with others. It is easy to verify from your submission whether you solved the problem on your own.
- **Questions.** If you have any requests or questions (technical difficulties, late submission due to inevitable circumstances, etc.), please ask the TAs on Blackboard. We generally encourage the use of Blackboard for discussions. However, for urgent or private issues, you can send an email to the responsible TA.

# References

[1] CSE251: System Programming. 2026. hw4.zip. https://websec-lab.github.io/courses/2026s-cse251/hw/hw4.zip.