CS-341L - Fall 2024
Cache Lab: Understanding Cache Memories

Assigned: November 11, 2024
Due dates: Part A - Design: November 18, 2024 at 11:59pm
Part A - Implementation: November 27, 2024 at 11:59pm
Part B: December 7, 2024 at 11:59 pm

1 Purpose

This project's main purpose is to understand the way the *cache* works in an example hardware architecture and how this knowledge can significantly improve program execution performance. Building a simulated cache manager in the C language, in addition to being more practice with C, gives you the opportunity to think through the data structures you need to represent the *cache* and think of the algorithms needed to show the cache's behavior. The C language is used in most low-level systems programs. Even if you will not be a systems administrator, but a developer, knowing how the cache works and having the experience of writing a simulator for it, will help your future professional development.

2 Logistics

This is an individual project. You may not co-develop the simulator with anyone else, but must design it and implement it on your own. The TA and the instructor may offer extra office hours if you need them. You must run this lab on a 64-bit x86-64 machine. Please start early. Make sure you read the whole statement of the assignment and understand it fully. For clarifications you can attend the office hours of your instructor or teaching assistant or you may send email to any of them, but ask for these clarifications early so that there is enough time to solve any problems before the deadline and so that you have plenty of time to invest in the project. This statement is 11 pages long, because we need to give you a lot of details. First read it completely and **start early**, you need all the allotted time.

3 Overview

This lab will help you understand the impact that cache memories can have on the performance of your C programs.

The lab consists of two parts. In the first part you will write a small C program (about 200-300 lines) that simulates the behavior of a cache memory. In the second part, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

4 Task

This section describes in great detail how to do the assignment. You need to download some files from the indicated directory in the CS machines. In addition to the skeleton of the two programs that you will build, compile, and run, which are called `csim.c` and `trans.c`, you will also get a trace file that serves as input to testing your cache simulator (see section 4.3).

4.1 Downloading the assignment

Log in to any of the cs machines. In the `/nfs/faculty/soraya/CS341L-F2024` directory in the cs machines you will find a file called `cachelab-handout.tar`, copy this file to a **protected (only you may read, write or execute) Linux directory** in which you plan to do your work.

Remember that that there are a couple of videos on canvas showing how to copy files and how to do sftp, you can go back to them if you are unsure of how to run those commands.

Then give the command:

```
linux> tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files. You will be modifying two files: `csim.c` and `trans.c`.

To compile these files, type:

```
linux> make clean
linux> make
```

WARNING: Do not let the Windows WinZip program open up your `.tar` file (many Web browsers are set to do this automatically). Instead, save the file to your Linux directory and use the Linux `tar` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files. Doing so can cause loss of data (and important work!).

4.2 Complete Description

In Part A of this project (described in section 4.4), you will implement a cache simulator. In Part B you will write a matrix transpose function that is optimized for cache performance (see section 4.5). First concentrate on Part A, but don't wait until a few days before the deadline (there will not be enough time). Part B is due only two weeks after, and then the final exam is the following week.

4.3 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

```
linux> 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.

NOTE: don't forget that the address values are given in hexadecimal.

4.4 Part A: Writing a Cache Simulator

The cache simulator that you will write for this part, is stored in `csim.c`, it 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.

This part A has two separate submissions, one for the design of the data structures and algorithms required to simulate the behavior of the cache, which is called *Design*. The second submissions is the actual implementation of this program, following your design. These two submissions are in separate assignments of this project.

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)

- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

The command-line arguments are based on the notation (s, E, and b) from pages 615-617 of the textbook (3rd edition). For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 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 - implementation is to fill in the `csim.c` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

This Part A - design is due first, then part A - implementation. See the submission details in section 7.

Programming Rules for Part A - implementation

- Include your name and netID in the header comment for `csim.c`.
- Your `csim.c` file must compile without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary s, E, and b. 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 lab, 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 - implementation, you must call the function `printSummary`, 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 project, 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.

4.5 Part B: Optimizing Matrix Transpose

In Part B you will write a transpose function in `trans.c` that causes as few cache misses as possible.

Let A denote a matrix, and A_{ij} denote the component on the ith row and jth 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 `trans.c` that computes the transpose of $N \times M$ matrix A and stores the results in $M \times N$ matrix B:

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(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.

You may use the provided simulator to solve the cache optimization problems in this part.

Part B is due last, see details in section 7.

Programming Rules for Part B

- Include your name and netID in the header comment for `trans.c`.
- Your code in `trans.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 to a single variable.

[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.

- Your transpose function may not use recursion.
- If you choose to use helper functions, you may not have more than 12 local variables on the stack at a time between your helper 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`.

5 Criteria for evaluation

This section describes how your work will be evaluated. The full score for this lab is 100 points, 40 points are for *Part A - design*, and 60 points are for *Part A - implementation* of the cache simulator and for *Part B* in which you optimize the transpose function. The distribution of the 60 points is as follows:

- Part A: 27 Points
- Part B: 26 Points
- Style: 7 Points

5.1 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters and traces. There are eight test cases, each worth 3 points, except for the last case, which is worth 6 points:

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

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 your reported number of hits, misses and evictions is worth 1/3 of the credit for that test case. That is, if a particular test case is worth 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points.

5.2 Evaluation for Part B

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

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

5.2.1 Performance (26 pts)

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 the reference simulator to replay this trace on a cache with parameters ($s = 5$, $E = 1$, $b = 5$).

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

- 32×32: 8 points if $m < 300$, 0 points if $m > 600$
- 64×64: 8 points if $m < 1,300$, 0 points if $m > 2,000$
- 61×67: 10 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.

5.3 Evaluation for Style

There are seven (7) points for coding style. These will be assigned manually by the course staff. We provide some Style guidelines in Canvas but the minimum you need to comply with is what we asked for in the first programming assignment: "Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive." The comments should be concise and appropriate to understand your code. Part A will get 4 points for style. Part B gets 3 points for style.

The course staff will inspect your code in Part B for illegal arrays and excessive local variables.

6 Working on the Lab

This section contains some details about each of the phases of this lab.

6.1 Working on Part A

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

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27
```

For each test, it shows the number of points you earned, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference 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 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 `csim.c` code, 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.

6.2 Working on Part B

We have provided you with an autograding program, called `test-trans.c`, 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 `trans.c` file. Each transpose version has the following form:

```
/* Header comment */
char trans_simple_desc[] = "A simple transpose";
void trans_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(trans_simple, trans_simple_desc);
```

in the `registerFunctions` routine in `trans.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 `trans.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 ($s = 5$, $E = 1$, $b = 5$).

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

```
linux> make
linux> ./test-trans -M 32 -N 32
Step 1: Evaluating registered transpose funcs for correctness:
func 0 (Transpose submission): correctness: 1
func 1 (Simple row-wise scan transpose): correctness: 1
func 2 (column-wise scan transpose): correctness: 1
func 3 (using a zig-zag access pattern): correctness: 1

Step 2: Generating memory traces for registered transpose funcs.

Step 3: Evaluating performance of registered transpose funcs (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151
func 2 (column-wise scan transpose): hits:870, misses:1183, evictions:1151
```

```
func 3 (using a zig-zag access pattern): hits:1076, misses:977, evictions:945

Summary for official submission (func 0): correctness=1 misses=287
```

In this example, we have registered four different transpose functions in `trans.c`. The `test-trans` 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 `test-trans` program saves the trace for function i in file `trace.f`i.[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:

  ```
  linux> ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f0
  S 68312c,1 miss
  L 683140,8 miss
  L 683124,4 hit
  L 683120,4 hit
  L 603124,4 miss eviction
  S 6431a0,4 miss
  ...
  ```

- 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.

- Blocking is a useful technique for reducing cache misses. See the following url

  `http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf`

  for more information.

6.3 Putting it all Together

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator and transpose code. This is the same program your instructor uses to evaluate your handins. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` 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:

```
linux> ./driver.py
```

[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.

7 Submitting Your Work

Part A - design is due on 11/18/2024. Upload a video describing your data structures and algorithms and a pdf of max. 3 pages to describe or visualize the structures and a high-level pseudocode for your algorithm.

Part A - implementation is due on 11/27/2024. Upload the file `csim.c` on Canvas by this due date. Your submission will be checked for plagiarism. Academic dishonest submissions will get zero (0) points for the whole lab and will be reported to the Dean of Students.

Each time you type `make` in the `cachelab-handout` directory, the Makefile creates a tarball, called `userid-handin.tar`, that contains your current `csim.c` and `trans.c` files.

Part B, consists of the `trans.c`. You will submit the .tar file for this final part **by the last deadline on 12/7/2024**. In addition to the .c file, you need to submit a short explanation of how you accomplished an improvement in the performance. If there were strategies that you tried and did not work, that is also relevant, especially if you do not obtain a big improvement. **The short explanations be will submitted in an** `EXPLAINED` **pdf file.** You will write a separate explanation for each matrix size.
You will submit your final `userid-handin.tar` **file on Canvas, in the place for that second portion specifically (it is a different assignment on Canvas).**

IMPORTANT: Do not create the handin tarball on a Windows or Mac machine, and do not handin files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files.