# 18-600, Fall 2018
# Cache Lab: Understanding Cache Memories
# Assigned: Sunday, October 28, 11:59PM PDT
# Due: Sunday, November 11, 11:59PM PDT
# Last Possible Time to Hand in: Wednesday, November 14, 11:59PM PDT

This is an individual lab assignment. All handins are done through AutoLab.

This lab assignment consists of three parts. The first part (Part A) involves writing a small C program (about 200-300 lines) that simulates the behavior of a cache for a single core processor. The second part (Part B) builds on the first part and involves extending the cache simulator to support private (per core) caches for a four-core processor by implementing the MSI cache coherence protocol among the four private caches. The last part (Part C) involves optimizing a small matrix transpose function with the goal of minimizing the number of cache misses.

## 1 Downloading the assignment

Your lab materials are contained in a Linux tar file called `cachelab-handout.tar`, which you can download from Autolab. Start by copying `cachelab-handout.tar` to a protected directory in Andrew in which you plan to do your work. Then login to a Linux machine and give the command

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

This will create a directory called `cachelab-handout` that contains a number of files. To compile these files for development, testing, etc., use the following commands:

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

Completing this assignment will involve modifications to these files:

- `csim.c` An LRU cache simulator

- `msim.c` An LRU cache simulator that supports the MSI coherence protocol for multi-threaded workloads
- `trans.c` A cache efficient matrix transpose implementation

In addition, we have also provided the skeleton files `cache.h` and `cache.c` to help you organize your code. Using these files is recommended but not mandatory.

**WARNING:** Don't expand the `.tar` file on your laptop. If you do, you might lose permission bits on some of the executable files. Instead, save the file to your AFS directory and use the Linux `tar` program to extract the files.

# 2 Overview

The lab has **three parts**. Parts A and B involve implementing a cache simulator for supporting the simulation of a cache for a single-core processor and the simulation of a multi-core processor with private caches and cache coherence, respectively. You may find it easier to complete Part A first, and then generalize this code for use in Part B. Part C involves writing a matrix transpose function that is optimized for cache performance.

## 2.1 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 100 points:

- Part A: 27 Points
- Part B: 36 Points
- Part C: 30 Points
- Coding Style: 7 Points

## 2.2 Evaluation for Style

There are 7 points for coding style. These will be assigned manually by the course staff. Style guidelines can be found on the course website. The course staff will inspect your code in Part C for illegal arrays and excessive local variables.

## 2.3 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 Parts A and B. The trace files are generated by a Linux program called `valgrind`.

The memory traces have the following form:

```
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one memory access. The format of each line is

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

The *operation* field denotes the type of memory access: “L” a data load, and “S” a data store. There is always a space before each “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.

# 3 Part A: Implement a Cache Simulator (27 points)

## 3.1 Description

In Part A you will write a cache simulator in `csim.c` that takes a 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 `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a 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 memory trace to replay

The command-line arguments are based on the notation ($s$, $E$, and $b$) from page 617 of the CS:APP3e textbook. 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
L 20,1 miss
S 20,1 hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
L 12,1 miss eviction
S 12,1 hit
hits:4 misses:5 evictions:3
```

Your job for Part A is to write a simulator in `csim.c` that takes the same command line arguments and produces the identical output as the reference simulator. Notice that `csim.c` contains a function `runSimulator()` where you should start your implementation.

### 3.2 Programming Rules for Part A

- Include your name and Andrew ID 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.
- To receive full credit for Part A, you must return the total number of hits, misses, and evictions, at the end of `runSimulator` function. See `struct sim_result` in `csim.h`.
- For this lab, 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 memory traces.

### 3.3 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters and traces. There are eight test cases (traces), each is 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.

### 3.4 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   21777   21745   265189   21777   21745  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:

- Your `csim.c` implementation is invoked from `csim-driver.c`. Feel free to take a look under the hood. This is where we do argument parsing and set up the traces.
- When implementing functions in `csim.c`, make sure to take a look at `csim.h` and `trace-stream.h` for more type information and documentation.
- Make use of `traceStreamNext` in `trace-stream.c` to read traces within `runSimulator`.
- 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.
- Each data load (L) or store (S) operation can cause at most one cache miss.

- Do not forget that the addresses in the trace are 64-bit hexadecimal memory addresses.
- Implementing a correct, stateful cache simulator in `cache.c` could save you lots of time when working on Part B.

# 4 Part B: Implement MSI Cache Coherence Protocol (36 points)

## 4.1 Description

In Part B, you will be writing a multi-cache simulator in `msim.c` for supporting a multi-core processor. Your work from Part A can be reused here, including how we define cache dimensions and the LRU replacement policy we have elected to use. However, there are some key differences:

- The `cache_config_t` for `msim.c` now contains multiple traces, one for each cache/core.
- Each trace represents a specific thread running on a specific core with its own local cache.
- Multiple threads running on the multiple cores produce the multiple traces of memory references.
- These traces can potentially access the same memory locations and therefore a cache coherence protocol must be implemented to ensure coherence among the multiple private caches.
- Caches now can also suffer coherence misses and the multi-cache simulator must report the number of `invalidations` misses triggered by the enforcement of cache coherence.

We have provided you with the binary executable of a reference cache simulator, called `msim-ref`. It can process up to four memory reference traces from four private caches, and simulate the MSI (Modified, Shared, Invalid) cache coherence protocol between the four private caches. Like Part A, it supports arbitrary cache dimensions and uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

### 4.1.1 Multi-Core Cache Organization

We now specify the cache organization you are to implement in Part B. This is important so that your simulator results will match those of `msim-ref` exactly.

- `msim` Can support the simulation of up to 4 simultaneous traces (from four threads running on four cores).
- Each core has its own private/local cache. All the private caches have the same implementation parameters, e.g. size, associativity, etc.
- Each cache implements the Least Recently Used (LRU) replacement policy within sets.
- Private caches are numbered 0-3 (corresponding to core numbering).

- Cache coherence between the four cores is maintained using the MSI protocol.
- Blocks/lines in each cache exist in one of three states: Modified, Shared, or Invalid.
- A *snooping* bus is used to allow the caches to communicate with each other and with main memory.
- The shared bus can only support a transaction from a single cache at a time. Therefore, broadcasts (e.g. memory write transactions) on the bus must be serialized (one after another).
- For the purpose of simulation, let updates from cache `C[i]` occur before updates from cache `C[i+1]`
- Also, assume that the sequencing of the memory operations between the cores is such that operation `op[j+1]` of cache `C[0]` occurs after operation `op[j]` of cache `C[max]`, i.e. round robin.

You may find it helpful when coding to pair various concepts above to functions and structs in your code. This bridges the semantic gap greatly, and will minimize your chances of error, or make troubleshooting more straightforward.

### 4.2 Programming Rules for Part B

- Include your name and Andrew ID in the header comment for `msim.c`.
- All rules from Part A still apply to `msim.c`.
- To receive full credit for Part B, you must return the total number of hits, misses, evictions, and invalidations at the end of `runSimulator` function. See `struct sim_results` in `msim.h`.
- Your `msim.c` implementation must support the simulation of up to four simultaneous traces.

### 4.3 Evaluation for Part B

TODO: Unlike Part A, you can invoke `msim` with multiple trace files. We will run the following test cases over your program with the appropriate point breakdown.

```
linux> ./msim -s 1 -E 1 -b 1 -t traces/yi2.trace # Still works!
linux> ./msim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./msim -s 5 -E 1 -b 5 -t traces/long.trace
linux> ./msim -s 4 -E 2 -b 4 -t traces/cc1.trace -t traces/cc2.trace
linux> ./msim -s 4 -E 2 -b 4 -t traces/cc1.trace -t traces/cc2.trace
-t traces/cc1.trace -t traces/cc2.trace
linux> ./msim -s 2 -E 4 -b 3 -t traces/trans.trace -t traces/trans.trace
linux> ./msim -s 5 -E 1 -b 5 -t traces/trans.trace -t traces/trans.trace
linux> ./msim -s 5 -E 1 -b 5 -t traces/long.trace -t traces/long.trace
```

### 4.4 Working on Part B

We have also provided you with an autograding program called `test-msim`. Be sure to compile your simulator before running the test cases:

```
linux> make
linux> ./test-msim
                                   Your simulator                Reference simulator
Points (s,E,b)      Hits Misses Evicts  Invalidations   Hits Misses Evicts  Invalidations
     4 (1,1,1)         9      8      6              0      9      8      6              0  -t traces/yi2.trace
     4 (4,2,4)         4      5      2              0      4      5      2              0  -t traces/yi.trace
     4 (5,1,5)    265189  21777  21745              0 265189  21777  21745              0  -t traces/long.trace
     4 (4,2,4)         2      4      0              2      2      4      0              2  -t traces/cc1.trace -t traces/cc2.trace
     4 (4,2,4)         6     14      0             10      6     14      0             10  -t traces/cc1.trace -t traces/cc2.trace
     -t traces/cc1.trace -t traces/cc2.trace
     4 (2,4,3)       326    150     10            109    326    150     10            109  -t traces/trans.trace -t traces/trans.trace
     4 (5,1,5)       345    131      0            119    345    131      0            119  -t traces/trans.trace -t traces/trans.trace
     8 (5,1,5)    487229   86703 27107          59534  87229  86703  27107          59534  -t traces/long.trace -t traces/long.trace
    36
```

For each test case, 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 B:

- Check to see how much of the code from Part A can be used in Part B.
- If your simulation results differ slightly, carefully read the multi-core cache organization specification above to gain insight on potential differences between your implementation and the reference implementation.
- In `cache.h` there is a function called `cacheBus` which should be implemented to perform the *snooping* function if you end up using `cache.h`.

## 5 Part C: Optimizing Matrix Transpose (30 points)

### 5.1 Description

In Part C you will implement a matrix transpose function in `trans.c` that consumes as few clock cycles as possible. Recall that cache misses incur significantly more clock cycles than cache hits.

Let $A$ denote a matrix, and $A_{ij}$ denote the element in the ith row and jth column of the matrix. 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 not performance efficient, because the memory access pattern results in relatively many cache misses, leading to a high number of clock cycles.

Your job in Part C is to write a similar function, called `transpose_submit`, that minimizes the number of clock cycles required for 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.

### 5.2 Programming Rules for Part C

- Include your name and Andrew ID 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.
- 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. Any attempt to use inline assembly will be interpreted as an effort to circumvent this restriction.
- 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`.

### 5.3 Evaluation for Part C

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

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

[1] The reason for this restriction is that we want you to limit your references to the stack and focus on the access patterns of the source and destination arrays.

### 5.3.1 Performance

For each matrix size, the performance of your `transpose_submit` function is evaluated by using `contech`-based instumentation[2] 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 clock cycles, $m$, up to some threshold. A cache miss incurs 100 clock cycles, while a cache hit incurs only 4 clock cycles. For example, a solution for the $32 \times 32$ matrix with 1800 hits and 300 misses ($m = (1800 * 4 + 300 * 100) = 37200$) would score the full 15 points.

- $32 \times 32$: 15 points if $m < 37,500$, 0 points if $m > 67,500$
- $64 \times 64$: 15 points if $m < 165,000$, 0 points if $m > 220,000$

Your code must be correct to receive any performance points for a particular size. Your code only needs to be correct for these two cases and you can optimize it specifically for these two 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.

## 5.4 Working on Part C

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);
```

[2] http://bprail.github.io/contech/

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

The autograder takes the matrix size as input. It uses `contech` 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 $32 \times 32$ 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
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255,
clock_cycles: 35700

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151,
clock_cycles: 121700

Summary for official submission (func 0): correctness=1 cycles=35700
```

In this example, we have registered two 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 C.

- The `test-trans` program saves the trace for function $i$ in file trace.f$i$. 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's performance, 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
  ...
  ```

- To debug the correctness of your transpose functions, you may need to invoke them directly using the executable `tracegen-ct`. `test-trans` executes `tracegen-ct` as part of the generating the traces; however, you can also execute this function directly. Please use the `-f` flag to specify which transpose function to use.

- 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. This will in turn lower the number of clock cycles required.
- Blocking is a useful technique for reducing cache misses as well. See

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

  for more information.

# 6 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 that Autolab uses when it autogrades your handins. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` to evaluate your submitted transpose function on the two matrix sizes. Then it prints a summary of your results and the points you have earned.

To run the driver, type:

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

# 7 Handing in Your Work

Each time you type `make` in the handout directory, it creates a tarball that contains your current `csim.c`, `msim.c`, `cache.c` and `trans.c` files.

To hand in your work for credit, run `make` on a shark machine to create the `andrewID_handin.tar` file, and then upload this tarball (and only this tarball!) to Autolab, which will autograde your submission and record your scores. You may handin as often as you like until the due date.

**IMPORTANT:** Do not create this file on a Windows or Mac machine, and do not upload files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files.