Cachelab

Due Dec 13, 2018 by 11:59pm **Points** 60 **Submitting** a file upload
File Types tar **Available** Dec 1, 2018 at 10am - Dec 14, 2018 at 11:59pm 14 days

This assignment was locked Dec 14, 2018 at 11:59pm.

For a properly formatted version of this assignment document, download either of these links:-- .**docx** **(https://canvas.wpi.edu/courses/10390/files/1619595/download?download_frd=1)** , .**pdf** **(https://canvas.wpi.edu/courses/10390/files/1619596/download?download_frd=1)** .

The Cachelab handout file is here:-- **cachelab-handout.tar** **(https://canvas.wpi.edu/courses/10390/files/1619572/download?download_frd=1)** (4 mBytes) or **cachelab-handout.tar.gz** **(https://canvas.wpi.edu/courses/10390/files/1619573/download?download_frd=1)** (100 kBytes)

Introduction

This lab is designed to help you to understand the impact that cache memories can have on the performance of *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. **This is the hard part!** In the second part, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

You may work on this assignment in *optional* two-person teams. If you and a classmate wish to work as a team, please send e-mail to **cs2011-staff@cs.wpi.edu (mailto:cs2011-staff@cs.wpi.edu)** as soon as possible, but *before December 7*. All submissions are electronic via *Canvas*. Either team member may submit on behalf of the team *under the team name*. Please put the names of both team members on *every* file that you modify.

> Academic Honesty Reminder:– Under the rules of this course, you may discuss you algorithms and approaches with your classmates and others, and you may consult on-line sources and other material for inspiration, but you may not copy code. Unauthorized copying will result in invoking the WPI procedures for academic dishonesty.
>
> If you have questions about what is or is not appropriate, please contact the Professor.

Logistics

We generated this lab using the –m64 flag of gcc, so that all code produced by the compiler follows x86-64 rules. This lab works on the virtual machines distributed for this course. It also does not work on the CCC Linux systems because of incompatible libraries, and it is unknown how it works on the

Macintosh.

You will need the valgrind program for this project. This has already been installed on the course virtual machines.

If, for some reason, **valgrind** is not installed on your virtual machine, open a command shell and execute the following command:–

sudo apt-get install valgrind

In addition, you should study carefully §§6.4 and 6.5 of the course textbook, *Computer Systems: A Programmer's Perspective*, 3rd edition, by Bryant and O'Hallaron.

Downloading the assignment

Download the lab from *Canvas* by following the link under the *Cachelab* assignment and selecting the file cachelab-handout.tar.gz. Save it to a protected Linux directory in which you plan to work. Once you have downloaded it, execute the following command in a shell in your Linux virtual machine:–

```
linux> tar xvzf cachelab-handout.tar.gz
```

This will create a directory called cachelab-handout containing a number of files. You will be modifying only two files:– csim.c and trans.c. To compile these files, type the following commands to a Linux command shell:–

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

Note:– The *only* files that you may modify in this assignment are csim.c and trans.c. Do *not* modify any other files. Modifying any other files will cause the autograding tools to fail and therefore will invalidate your submission of this assignment.

Part A: Cache Simulator

In this part of the assignment, you will write a program in *C* to simulate a processor cache. The simulator will be parameterized by the number of sets, the degree of associativity, and the size of blocks in the cache. It will accept as input *trace files* containing lists of memory references. It must mimic the cache hits, misses, and evictions of the blocks corresponding to the addresses in those traces, and it must report the performance of the cache.

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 (see below). The trace files are generated by a program called valgrind. For example, typing the command

linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l

causes valgrind to run the executable program “ls -l”, capturing a trace of its memory accesses in the order they occur, and prints them on stdout.

Memory traces created by valgrind have the following form (note spacing and indentation):–

```
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 hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation.

Writing a Cache Simulator

In Part A you will write a cache simulator in csim.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 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 command has the following arguments:

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

where the arguments are:–

- -h: Optional help flag that prints usage info
- -v: Optional verbose flag that displays trace info
- -s <s>: Number of set index bits (the number of sets is 2^s)
- -E <E>: Associativity (number of lines per set)
- -b <b>: Number of block bits (the block size is 2^b)

- -t <tracefile>: Name of the valgrind trace to replay

The command-line arguments are based on the notation (*s*, *E*, and *b*) from §6.4.1, page 615, of the course 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
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 the csim.c file so that it takes the same command line arguments and produces the identical output as the reference simulator (csim-ref). Notice that csim.c is almost completely empty. *You must write it from scratch.*

Programming Rules for Part A

- Include your name and your WPI username in a comment line at the top of c. If you are a team, provide the usernames of both team members, as registered in *Canvas*. For example, if your team members are smith2 and jones4, then the team name is jones4-smith2.
- Your c file must compile without warnings in order to receive credit. (The –Wall flag is automatically applied in the makefile.)
- Your simulator must work correctly for arbitrary values of *s*, *E*, and *b*. This means that you will need to allocate storage for your simulator's data structures using the malloc Type "man malloc" to a Linux command shell 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, the main function of your csim program must call the function printSummary(), with the total number of hits, misses, and evictions, at the end of your simulation:–

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

- For this 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 valgrind

Caution: Do not include *any* non-standard header files (i.e., .h files) in your csim.c or trans.c programs, other than those already included.**[1]** The autograder will not find any other headers, and you will get zero points for that part of the assignment.

Part B: Matrix Transpose

In Part B you will write a transpose function in trans.c that minimizes cache misses to as few 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 a lot of 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”) of the transpose_submit function. The autograder searches for this string to determine which transpose function to evaluate for credit.

Programming Rules for Part B

- Include your name(s) and WPI username in a comment line at the top of c. If you are working as a team, include the usernames of both team members as indicated above (in Part A).
- Your code in c must compile without warnings to receive credit.
- You are allowed to define at most 12 local variables of type int per transpose function.**[2]**
- 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.

- 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.

Note about Part B:– Students in previous terms have noted that make often takes an un-usually long time for this part. This has been attributed to valgrind. In particular, valgrind has been known to have trouble analyzing and interpreting linked lists. Therefore, we *strongly suggest* that you avoid using linked lists in your implementation.

Evaluation

This section describes how your work will be evaluated. The full score for this lab is 60 points, allocated as follows:–

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

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. Be sure to use gdb or another debugger to help understand what your programs are doing.

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.

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)
- A size to be chosen after the assignment is closed. For testing purposes, the autograder distributed as part of cachelab-handout uses *M = 61, N = 67*. The size of the matrix used for grading will be approximately 4096 entries, and the row size may be a prime or other odd number.

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
- other size: 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.

Evaluation for Style

There are 7 points for coding style. These will be assigned manually by the course staff. The course staff will also inspect your code in Part B for illegal arrays and excessive local variables. Violations will result in deductions from the Performance points.

Working on the Lab

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	(Total)							

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 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.

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 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 fi.**[3]** 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

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

for more information.

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 similar to the program that your instructor uses to evaluate your submissions (except for the *61×67* matrix). 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
```

Submitting your work

Each time you type make in the cachelab-handout directory, the Makefile creates a tarball, called username-handin.tar, that contains your current csim.c and trans.c files. In the file name of this tarball, “username” is replaced by the user name of the person running the make command. For grading purposes, this *must* be your WPI username.

If you are working as a team, your file must be named teamname-handin.tar, where teamname is replaced by your team name as registered in *Canvas.*

Whether working as an individual or in a team, the best and most reliable way of creating a correctly named tarball that works with the autograder is to use the command

> make USER=teamname or make USER=username

Submit your tarball to *Canvas under project Cachelab.*

> IMPORTANT: Do not create the tarball on a Windows or Mac machine, and do not submit files in any other archive format, such as .zip, .gzip, or .tgz files. The autograding programs will be looking specifically for tarballs with names as specified and in a format created by the make command of this assignment

Extra Credit

For 15 points of extra credit, submit a tarball containing a working version of csim.c by 6:00 PM, Saturday, December 8, 2018. The extra credit submission *must be named* extra-username-handin.tar or extra-teamname-handin.tar. You may modify csim.c after you submit the extra credit.

Note that in order to earn the extra credit, the autograder must be able to build and run the submitted version of csim.c.

[1] Of course, you may include and use standard headers such as <stdlib.h> and <string.h>.

[2] The reason for this restriction is that the autograder testing code is not able to count references to the stack. You should limit your references to the stack and focus on the access patterns of the source and destination arrays.

[3] 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.

Cachelab Rubric

Criteria	Ratings	Pts
Autograder Score Score from autograder		53 pts
Style points		7 pts
Extra Credit Tarball containing a working version of csim.c by 6:00 PM, Saturday, December 8, 2018		15 pts
		Total Points: 75