天津大学

计算机系统基础上机实验报告

实验题目 5：高速缓存 cache

学院名称 智能与计算学部

专　　业 工科试验班

学生姓名 张明君

学　　号 6319000359

年　　级 2019 级

班　　级 留学生班

时　　间 2020 年 6 月 15 日

实验 5：高速缓存

Cache

1. 实验目的

进一步理解高速缓存对于程序性能的影响。

2. 实验内容

这个实验包括以下两部分内容：你需要使用 C 语言编写一个小型程序（200-300 行）用来模拟高速缓存； 然后，对一个矩阵转置函数进行优化，以减少函数操作中的缓存未命中次数。详细内容请参考实验指导书：实验 5.pdf

No.	任务内容
1	任务 A：编写一个高速缓存模拟程序。在这部分任务中，你将在 csim.c 文件中编写一个高速缓存仿真程序。这个程序使用 valgrind 的内存跟踪记录 作为输入，模拟高速缓存的命中/未命中行为，然后输出总的命中次数，未命中次数和缓存块的替换次数。
2	任务 B：优化矩阵转置运算程序。在 trans.c 中编写一个矩阵转置函数，尽可能的减少程序对高速缓存访问的未命中次数。
3	

3. 实验要求

1） 在 Unbuntu18.04LTS 操作系统下，按照实验指导说明书，使用 gcc、make 和内存访问进行捕获和追踪的工具，完成本实验。

2） 本实验的具体要求：

a) 编译时不允许出现任何的 warning。

b) 转置函数中定义的 int 型局部变量总数不能超过 12 个。

c) 不允许使用 long 等数据类型，在一个变量中存储多个数组元素以减少内存访问。

d) 不允许使用递归。

e) 在程序中不能修改矩阵 A 中的内容，但是，你可以任意使用矩阵 B 中的空间，只要保证终的结果正确 即可。

f) 在函数中不能定义任何的数组，不能使用 malloc 分配额外的空间。

3） 需提交：csim.c 和 trans.c 源文件，电子版实验报告全文。

4. 实验结果

- Part A

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.

```
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

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. Notice that this file is almost completely empty. You'll need to write it from scratch.

- Include your name and loginID 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, 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 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` traces.

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.

- 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.trac
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trac
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trac
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trac
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
```

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.

- Part B

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.

- 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.
- 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`.

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 ($M = 61$, $N = 67$)

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.

- 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 reg- istered 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 trans- pose 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.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:

```
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.

- Put all the files 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 TA 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

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