CacheLab 实验报告

李国祥 2018202135

实验目的

本实验主要通过用 C 语言实现一个高速缓存模拟器(Cache Simulator)，以及优化矩阵转置，减少矩阵转置中的缓存 Miss 数来达到优化的目的，来更进一步地熟悉 cache.

Part A

完成源文件 *csim. c*，实现 Cache Simulator. 查看实验说明，这里要让 *csim. c* 编译出来的 test-csim 和 csim-ref 具有相同的功能.

1. 接受一个命令行参数，从中解析出 cache 的参数(s, E, b)，以及要读取的地址序列所存在的文件，以及辅助功能(打印帮助信息，以及在顺序读取文件时打印每一次读取是 Miss, Hit 还是 Evict)，这里由于不测试辅助功能，所以只实现了前面的功能. 命令行参数的解析用 getopt 函数实现, getopt 的具体用法用命令 man getopt 查看即可.
2. trace 操作字段的解释:L 表示数据加载，即读. S 表示数据储存，即写. M 表示数据修改，即读和写. L, S 都只需读一次 Cache. M 需要读两次. 这里用 fscanf 从文件中读指令和地址.
3. 实现一个 cache，并且完成 cache 读取时的 LRU(Least Recently Used)策略.
   为实现 cache 定义了以下几个结构体:

`typedef struct`
`{ // a cacheline`
`int time_stamp;`
`int valid;`
`int tag;`
`} cache_line;`
	即高速缓存块，三个变量分别是时间戳，有效位和标志位. 时间戳用于帮助实现 LRU 策略. 正常的缓存还要实现储存内容的 block, 这里由于只是通过不断访问对 Miss, Hit, Evict 计数，故 block 可以不用实现.

`typedef struct`
`{ // a cache set has many lines`
`cache_line *lines;`
`} cache_set;`
	即缓存组，一个组有多个行，在这里即描述为缓存组是一个缓存行的数组.

`typedef struct`
`{ // a cache has many sets`
`cache_set *sets;`
`} cache;`
	即缓存，一个缓存有多个组，在这里描述为缓存是一个缓存组的数组. 即储存缓存行的二维数组.

`typedef struct`
`{`
`int s;`
`int b;`
`int E;`
`int t;`
`int hits;`
`int misses;`
`int evicts;`
`} cache_ctrl;`
	储存缓存的一些信息，以及记录 Hit, Miss, Evict 数.

以及以下几个函数:

```
cache Init(int number_of_sets, int number_of_lines)
```

用缓存组数(S)和每组行数(E)来初始化一个 cache，函数中主要使用了 malloc 函数来为 cache 分配给定的空间.

```
void Free_Cache(cache *MyCache, cache_ctrl Panel)
```

cache 销毁的函数. 由于初始化的时候用了 malloc，函数中用 free 来回收内存.

```
void Address_Parser(cache_ctrl Panel, Address address,
Address *tag, Address *set_index)
```

地址解析函数. Address 是自己定义的 unsigned long long 类型，这个函数从一个地址中解析出组索引和标志位.

```
void Cache_Read(cache MyCache, cache_ctrl *Panel,
Address address)
```

最核心的函数，模拟用一个地址读取 cache. 读取具有以下逻辑:

1) 遍历地址解析出来的组索引对应的组，如果在发现空行前找到了和地址具有相同标志位的行，则得到一次 Hit. 这一行时间戳+1.
2) 发现空行，得到一次冷不命中 Miss. 把地址放入这个空行，这个空行的时间戳为当前组所有行时间戳的最大值+1.
3) 没有空行，得到一次冲突不命中 Miss. 执行 LRU 策略，将当前组所有行时间戳最小的行替换成新的地址，得到一次 Evict，将被替换的行的时间戳设为当前组所有行时间戳的最大值+1.

如上即可实现 Cache Simulator. make 编译, . /test-csim 测试, Part A 完成，更多细节请看代码.

测试完成截图

```
2018202135@VM-0-46-ubuntu:~/cachelab-handout$ ./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


TEST_CSIM_RESULTS=27
```

Part B

完成源文件 *trans. c*，实现对 3 个矩阵转置算法的优化以减少 Miss 数. 矩阵转置的函数原型:

```
void trans(int M, int N, int A[N][M], int B[M][N])
```

矩阵 B 是目标矩阵.

优化的目标如下:

32×32 矩阵, $misses < 300$.

64×64 矩阵, $misses < 1300$.

61×67 矩阵, $misses < 2000$.

如果直接转置:

```
for(int i = 0; i < N; i ++)
    for(int j = 0; j < M; j ++)
        B[j][i] = A[i][j];
```

对应的 Miss 数为:

32×32 矩阵, $misses = 1187$.

64×64 矩阵, $misses = 4723$.

61×67 矩阵, $misses = 4423$.

32×32矩阵

先分析直接转置 Hiss, Miss 和 Evict 的情况.

用. /test-trans –M 32 –N 32 命令测试对32×32矩阵的转置, 然后看生成的 trace. f1(直接转置函数生成的)文件. 即是一系列按地址的访问. 用. /csim-ref –s 5 –E 1 –b 5 –v –t trace. f1, 查看每一次地址访问的情况.

trace. f1:

```
 1  S 0038b08c, 1
 2  L 0038b0c0, 8
 3  L 0038b084, 4
 4  L 0038b080, 4
 5  L 0030b080, 4
 6  S 0034b080, 4
 7  L 0030b084, 4
 8  S 0034b100, 4
 9  L 0030b088, 4
10  S 0034b180, 4
```

从标红的两位数, 以及第 5 行后 L, S 指令的交替出现可以推测从第 5 行开始函数开始访问数组 A 和 B. L 是对 A 的读访问, S 是对 B 的写访问.

. /csim-ref –s 5 –E 1 –b 5 –v –t trace. f1 的结果:

```
1  S 38b08c, 1 miss
2  L 38b0c0, 8 miss
3  L 38b084, 4 hit
4  L 38b080, 4 hit
5  L 30b080, 4 miss eviction
6  S 34b080, 4 miss eviction
7  L 30b084, 4 miss eviction
8  S 34b100, 4 miss
9  L 30b088, 4 hit
```

```
10 | S 34b180, 4 miss
```

将第 5 行开始，对矩阵访问的 Hit, Miss 和 Evict 情况绘成表格，如下. *M* 代表 miss eviction, *m* 代表 miss, *h* 代表 hit

矩阵 A:

M	*M*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*m*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*M*	*h*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*M*	*h*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*M*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*M*	*h*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*M*	*h*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*M*	*h*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*M*
M	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*M*	*h*	*h*	*h*	*h*	*h*	*h*	*h*

四列的 Miss 是好理解的，一个 cache 有 32 个字节，刚好放下 8 个 int. 对角线向上一格的 Miss，是因为之前元素在对角线上, B 矩阵的元素地址和 A 矩阵的元素地址解析出来的组索引相同，对于矩阵的同一个位置，先读 A，再写 B，所以 B 将对应的 A 的块驱逐了. 之后又读 A 的下一个元素，故出现了上表的情况.

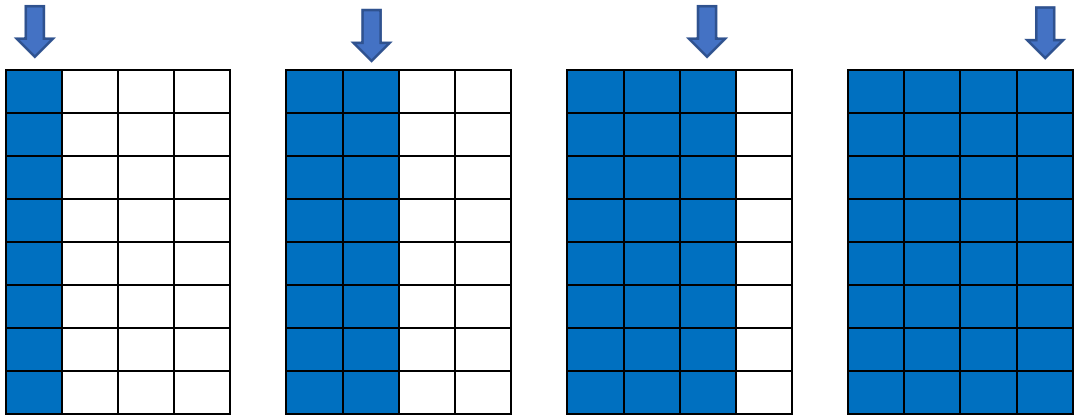
B 矩阵的情况:

M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
m	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

无一例外，对 B 的访问全 Miss 了. 分析原因，当横着从左往右读 A 的时候，一直是从上往下竖着写 B，所以任意两次写 B 之间有 32 个 int，超过了一个 cache 中的 8 个 int. 而$S = s^5 = 32$，即 cache 中最多存 32 组，所以访问 B 完全没有利用到 cache.

所以优化的方向也可以由此确定了，让对 B 的写访问可以更好地利用 cache.

这里的方法是一次写 B 的 8 行，从左往右写:



可以确保上下两行不会映射到同一个组内，这样除第一列外，之后的写访问不会再造成 Miss. (对角线上的元素除外，对角线元素造成 Miss 的原因前面已经解释过了)，同时对 A 访问造成的 Miss 不会增多(实际上由于一次读 A 的 8 个元素，之前所说由于访问对角线位置的 B 造成访问 A 对角线上方元素的 Miss 也不会产生了).

按照这个思路写出的代码:

```
 for (j = 0; j < 32; j += 8)
 {
    for (i = 0; i < 32; i++)
     {  // i 是内层循环，故矩阵 B 的每 8 行是从左往右写的
        x1 = A[i][0 + j];
        x2 = A[i][1 + j];
        x3 = A[i][2 + j];
        x4 = A[i][3 + j];
        x5 = A[i][4 + j];
        x6 = A[i][5 + j];
        x7 = A[i][6 + j];
        x8 = A[i][7 + j];
        B[0 + j][i] = x1;
        B[1 + j][i] = x2;
        B[2 + j][i] = x3;
        B[3 + j][i] = x4;
        B[4 + j][i] = x5;
        B[5 + j][i] = x6;
        B[6 + j][i] = x7;
        B[7 + j][i] = x8;
    }
}
```

make 编译，. /test-trans –M 32 –N 32，得到如下结果:

```
2018202135@VM-0-46-ubuntu:~/cachelab-handout$ ./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
```

Miss 数已经降到 287，符合要求.

看优化后访问矩阵 B 的情况:

M h h h h h h M h h h h h h h m h h h h h h m h h h h h h h
m M h h h h h h m h h h h h h h m h h h h h h m h h h h h h h
m h M h h h h h m h h h h h h h m h h h h h h m h h h h h h h
m h h M h h h h m h h h h h h h m h h h h h h m h h h h h h h
m h h h M h h h m h h h h h h h m h h h h h h m h h h h h h h
m h h h h M h h m h h h h h h h m h h h h h h m h h h h h h h
m h h h h h M h m h h h h h h h m h h h h h h m h h h h h h h
m h h h h h h M m h h h h h h h m h h h h h h m h h h h h h h
M h h h h h h h M h h h h h h h M h h h h h h M h h h h h h h
M h h h h h h h M M h h h h h h M h h h h h h M h h h h h h h
M h h h h h h h M h M h h h h h M h h h h h h M h h h h h h h
M h h h h h h h M h h M h h h h M h h h h h h M h h h h h h h
M h h h h h h h M h h h M h h h M h h h h h h M h h h h h h h
M h h h h h h h M h h h h M h h M h h h h h h M h h h h h h h
M h h h h h h h M h h h h h M h M h h h h h h M h h h h h h h
M h h h h h h h M h h h h h h M M h h h h h h M h h h h h h h
M h h h h h h h M h h h h h h h M h M h h h h h M h h h h h h
M h h h h h h h M h h h h h h h M h h M h h h h M h h h h h h
M h h h h h h h M h h h h h h h M h h h M h h h M h h h h h h
M h h h h h h h M h h h h h h h M h h h h M h h M h h h h h h
M h h h h h h h M h h h h h h h M h h h h h M h M h h h h h h
M h h h h h h h M h h h h h h h M h h h h h h M M h h h h h h
M h h h h h h h M h h h h h h h M h h h h h h h M h h h h h h
M h h h h h h h M h h h h h h h M h h h h h h h M M h h h h h
M h h h h h h h M h h h h h h h M h h h h h h h M h M h h h h
M h h h h h h h M h h h h h h h M h h h h h h h M h h M h h h
M h h h h h h h M h h h h h h h M h h h h h h h M h h h M h h
M h h h h h h h M h h h h h h h M h h h h h h h M h h h h M h
M h h h h h h h M h h h h h h h M h h h h h h h M h h h h h M
M h h h h h h h M h h h h h h h M h h h h h h h M h h h h h h

和之前的推测是一样的，对角线上的 Miss 是由于访问相同位置的 A 造成的.

64 × 64 矩阵

这可以说是 Part B 的难点，首先64 × 64矩阵刚好就是32 × 32矩阵的 4 倍大小，那么用上面的方法可不可以呢?

把上面循环的两个 32 改成 64，刚好也实现了转置的功能.

结果:

```
2018202135@VM-0-46-ubuntu:~/cachelab-handout$ ./test-trans -M 64 -N 64

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:3586, misses:4611, evictions:4579
```

Miss 数 4611，和直接转置几乎没有区别. 距离目标 1600 差距很大.

对矩阵 A 的访问肯定不是 Miss 的主要来源，还是考虑矩阵 B. B 上下相邻两个元素地址相差$64 * \text{sizeof(int)} = 256 = 0\text{x}100$.

$s = 5, b = 5$. 所以 B 上下相邻两个元素的组索引只会有两位(最靠前的两位)是不相同的. 这意味着一次访问的 8 个 B 中元素只能映射到$2^2 = 4$个组当中. B[4+i][j]和 B[i][j]的组索引是一样的.

用 csim-ref 验证这个想法:查看连续两次访问 B 中的一列 8 个元素:

S 34b080, 4 miss eviction (B[0][0])	S 34b084, 4 miss eviction (B[0][1])
S 34b180, 4 miss (B[1][0])	S 34b184, 4 miss eviction (B[1][1])
S 34b280, 4 miss (B[2][0])	S 34b284, 4 miss eviction (B[2][1])
S 34b380, 4 miss (B[3][0])	S 34b384, 4 miss eviction (B[3][1])
S 34b480, 4 miss eviction (B[4][0])	S 34b484, 4 miss eviction (B[4][1])
S 34b580, 4 miss eviction (B[5][0])	S 34b584, 4 miss eviction (B[5][1])
S 34b680, 4 miss eviction (B[6][0])	S 34b684, 4 miss eviction (B[6][1])
S 34b780, 4 miss eviction (B[7][0])	S 34b784, 4 miss eviction (B[7][1])

确实是全 Miss 了. 而且 B[0][0]的地址为 0x34b080, B[4][0]的地址为 0x34b480. 转换成二进制，后 10 位即完全一样，结果就是组索引相同.

所以一次 8 个转置是不行了，而上面讨论的结果给了一定的启发:既然 B 数组上下最多 4 个元素才能映射到不同的组当中，那可不可以调整步长为 4，即一次 4 个转置呢?

只需要对上面32×32矩阵的代码做少许修改:

```
for (j = 0; j < 64; j += 4)
{
    for (i = 0; i < 64; i++)
    {
        x1 = A[i][0 + j];
        x2 = A[i][1 + j];
        x3 = A[i][2 + j];
        x4 = A[i][3 + j];
        // x5 = A[i][4 + j];
        // x6 = A[i][5 + j];
        // x7 = A[i][6 + j];
        // x8 = A[i][7 + j];
        B[0 + j][i] = x1;
        B[1 + j][i] = x2;
        B[2 + j][i] = x3;
```

```
        B[3 + j][i] = x4;
        // B[4 + j][i] = x5;
        // B[5 + j][i] = x6;
        // B[6 + j][i] = x7;
        // B[7 + j][i] = x8;
    }
}
```

结果：

```
20182021350@VM-0-46-ubuntu:~/cachelab-handout$ ./test-trans -M 64 -N 64

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:6546, misses:1651, evictions:1619
```

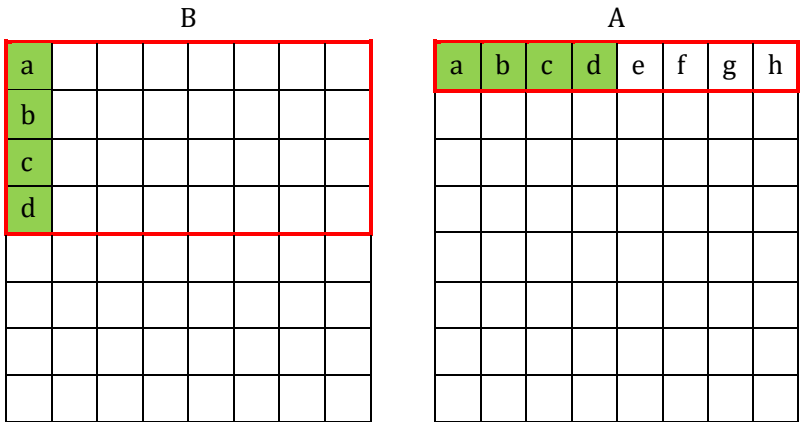
Miss 数变为 1651，但和 1300 的要求仍有差距.

还有没有别的办法呢?之前选择一次转置 8 个是因为 cache 一个组是 32 个字节，能存 8 个 int. 这里不能一次转置 8 个，一次最多转置 4 个，但 cache 还是会一次把 A 矩阵的 8 个数都读进同一个组. 上面的代码只用了前面 4 个数，后面 4 个数被浪费了(在下一轮循环时被逐出，等到需要转置这 4 个数的时候，这 4 个数又被加载进 cache). 如果能利用好这 4 个数，应当可以进一步优化.

这 4 个数应当放在某些位置，但如果用临时变量存是不够的. 如果要用临时变量存，则需要$4 \times 64 = 256$个临时变量(在这 4 个数真正派上用场之前还要经历 63 轮循环，所以会产生这么多需要临时储存的变量)，超出了限制，编程也会比较复杂.

所以只能考虑把他们放到矩阵 B 的某些位置. 下面即是一种基于此想法的实现.

(a) 一次读 a, b, c, d，把它们放到 B 数组的相应位置，同时 e, f, g, h 被载入缓存, B 中修改的 4 行也载入缓存. 需要注意的是它们映射到了不同的组中.



(b) 将 e, f, g, h 放到 B 如图所示的位置.

B

a				e			
b				f			
c				g			
d				h			

A

a	b	c	d	e	f	g	h

(c) 对 A 的前四行重复(a)(b)步骤. 结果如下图:

B

a	i	q	y	e	m	u	C
b	j	r	z	f	n	v	D
c	k	s	A	g	o	w	E
d	l	t	B	h	p	x	F

A

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	A	B	C	D	E	F

值得注意的是, 在此过程中, B 中的缓存是得到充分利用了的. 除第一列外, 之后的写操作都是 Hit. A 矩阵显然也充分利用了 cache.

(d) 一次将 e, m, u, C 读出来, 用临时变量存起来. 同时将 A 中 G, P, X, 5 部分转置到这四个位置, 注意此时 cache 存了 A 的下半部分, 上半部分被驱逐. B 部分 cache 没有发生变化.

B

a	i	q	y	G	P	X	5
b	j	r	z	f	N	v	D
c	k	s	A	g	o	w	E
d	l	t	B	h	p	x	F
e	m	u	C				

A

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	A	B	C	D	E	F
G	H	I	J	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	0	1	2	3	4
5	6	7	8	9	~	!	@

(e) 将 e, m, u, C 写入转置后的位置，同时注意 B 第 5 行把第 1 行从 cache 中驱逐了.

B

a	i	q	y	G	P	X	5
b	j	r	z	f	N	v	D
c	k	s	A	g	o	w	E
d	l	t	B	h	p	x	F
e	m	u	C				

A

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	A	B	C	D	E	F
G	H	I	J	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	0	1	2	3	4
5	6	7	8	9	~	!	@

(f) 重复(d)(e)操作，将 B 的右上4 × 4区域和 A 的左下4 × 4区域转置到位:

B

a	i	q	y	G	P	X	5
b	j	r	z	H	Q	Y	6
c	k	s	A	I	R	Z	7
d	l	t	B	J	S	0	8
e	m	u	C				
f	N	v	D				
g	o	w	E				
h	p	x	F				

A

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	A	B	C	D	E	F
G	H	I	J	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	0	1	2	3	4
5	6	7	8	9	~	!	@

这部分操作, A 的 cache 已经不再发生变化，而 B 的 cache 可以描述成平滑向下移动的过程，不会出现上下跳动的情况.

(g) 现在只剩右下的4 × 4区域，可以发现它们都在 cache 中，所以直接转置即可.

B

a	i	q	y	G	P	X	5
b	j	r	z	H	Q	Y	6
c	k	s	A	I	R	Z	7
d	l	t	B	J	S	0	8
e	m	u	C	L	T	1	9
f	N	v	D	M	U	2	~
g	o	w	E	N	V	3	!
h	p	x	F	O	W	4	@

A

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	A	B	C	D	E	F
G	H	I	J	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	0	1	2	3	4
5	6	7	8	9	~	!	@

至此这样8 × 8转置结束，将64 × 64矩阵分成 64 个这样的8 × 8的块，依次转置，即可完成对整个矩阵的转置.

按照上面的思路实现的代码:

```
for (i = 0; i < N; i += 8)
    for (j = 0; j < M; j += 8)
        { // 对应 64 个 8*8 的分块
            for (k = 0; k < 4; k++)
            { // 对应步骤(a)(b)(c)
                x1 = A[i + k][j + 0];
                x2 = A[i + k][j + 1];
                x3 = A[i + k][j + 2];
                x4 = A[i + k][j + 3];
                x5 = A[i + k][j + 4];
                x6 = A[i + k][j + 5];
                x7 = A[i + k][j + 6];
                x8 = A[i + k][j + 7];

                B[j + 0][i + k] = x1;
                B[j + 1][i + k] = x2;
                B[j + 2][i + k] = x3;
                B[j + 3][i + k] = x4;

                B[j + 0][i + k + 4] = x5;
                B[j + 1][i + k + 4] = x6;
                B[j + 2][i + k + 4] = x7;
                B[j + 3][i + k + 4] = x8;
            }
             for (k = 0; k < 4; k++)
             { // 对应步骤(e)(f)(g)
                x1 = B[j + k][i + 4];
                x2 = B[j + k][i + 5];
                x3 = B[j + k][i + 6];
                x4 = B[j + k][i + 7];

                x5 = A[i + 4][j + k];
                x6 = A[i + 5][j + k];
                x7 = A[i + 6][j + k];
                x8 = A[i + 7][j + k];

                B[j + k][i + 4] = x5;
                B[j + k][i + 5] = x6;
                B[j + k][i + 6] = x7;
                B[j + k][i + 7] = x8;

                B[j + 4 + k][i + 0] = x1;
                B[j + 4 + k][i + 1] = x2;
```

```
            B[j + 4 + k][i + 2] = x3;
            B[j + 4 + k][i + 3] = x4;
        }
        for (k = 4; k < 8; k++)
        { // 对应步骤(g)
            for (l = 4; l < 8; l++)
            {
                B[j + l][i + k] = A[i + k][j + l];
            }
        }
```

测试结果如下:

```
2018202135@VM-0-46-ubuntu:~/cachelab-handout$ ./test-trans -M 64 -N 64

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:8970, misses:1275, evictions:1243
```

Miss 数成功减少为 1275 次, 达到了标准.

61 × 67矩阵

看上去没有64 × 64矩阵规则. 但实际上, 这样出现分块后几行映射到同样的几个组的情况更不容易发生了, 而且要求比64 × 64矩阵更松, 只要 Miss 数减少到 2000 即可. 这里尝试了一下在56 × 64的区域一次转置 8 个, 在剩余 56 × 3的区域一次转置 8 个, 再在剩余的5 × 67的区域一次转置 5 个.

		61	
		56	5
67	64	一次转置 8 个	一次转置 5 个
	3	一次转置 8 个	

得到结果如下:

```
2018202135@VM-0-46-ubuntu:~/cachelab-handout$ ./test-trans -M 61 -N 67

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:6186, misses:1993, evictions:1961
```

1993 的 Miss 数, 达到要求.

总结 Part B，核心思想是一块一块地移动数据，不要让同一部分数据加载进缓存，还没用到又被驱逐，等这部分数据需要使用的时候又将其加载进 cache. 64×64 矩阵转置的优化是最具有技巧性的.

make 编译，. /driver. py 得到总的结果，如下图:

```
20182021135@VM-0-46-ubuntu:~/cachelab-handout$ ./driver.py
Part A: Testing cache simulator       53
Running ./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


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1275
Trans perf 61x67          10.0        10        1993
          Total points    53.0        53
```

cachelab 到这里就完成了.