哈爾濱工業大學

实验报告

实 验（六）

题　　目 Cachelab

高速缓冲器模拟

专　　业 计算学部

学　　号 1190200717

班　　级 1903008

学　　生 梁浩

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021/6/3

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统存储器层级结构
掌握 Cache 的功能结构与访问控制策略
培养 Linux 下的性能测试方法与技巧
深入理解 Cache 组成结构对 C 程序性能的影响

1.2 实验环境与工具

1.2.1 硬件环境

处理器：Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40GHz
已安装的内存(RAM)：8.00GB(7.81GB 可用)
系统类型：64 位操作系统，基于 x64 的处理器

1.2.2 软件环境

Windows 10 家庭中文版; VirtualBox 6.1; Ubuntu 20.04

1.2.3 开发工具

visual studio, Valgrind, Nodepad++

1.3 实验预习

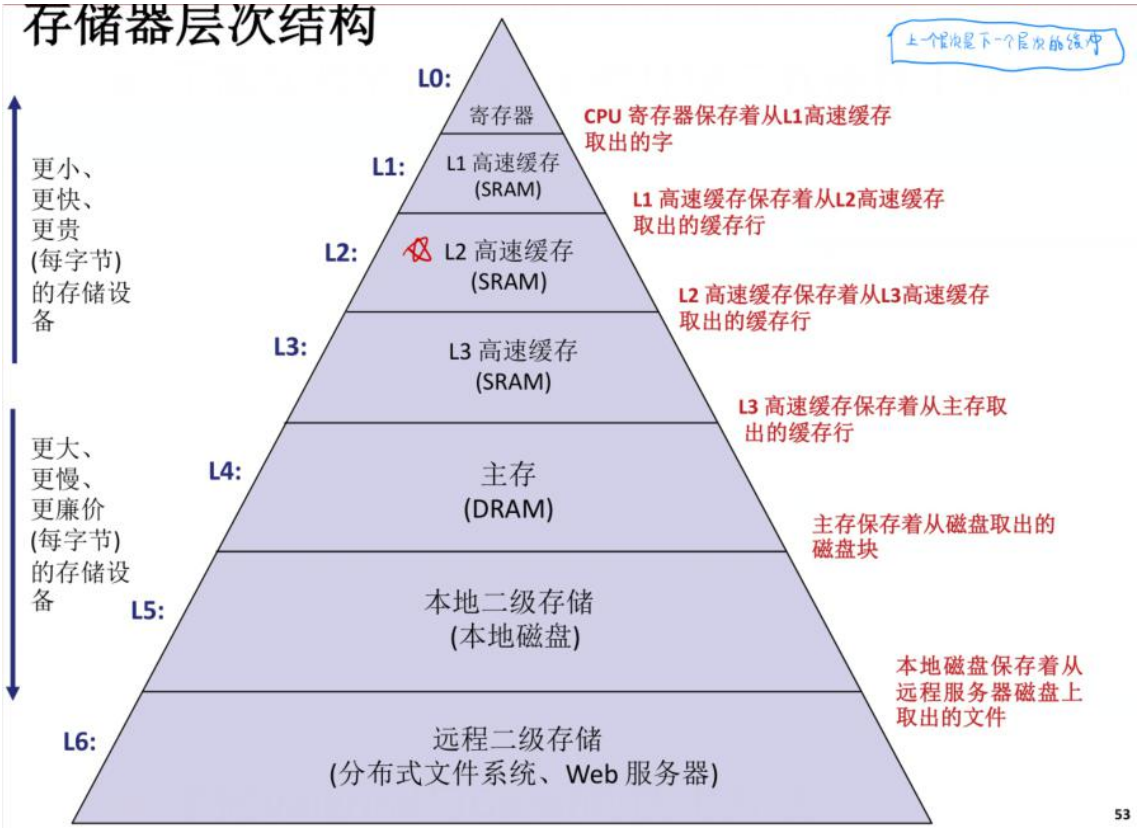
上实验课前，必须认真预习实验指导书（PPT 或 PDF）
了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。
画出存储器的层级结构，标识其容量价格速度等指标变化
用 CPUZ 等查看你的计算机 Cache 各参数，写出 C S E B s e b 缓存大小 C、分组数量 S、关联度/组内行数 E、块大小 B，及对应的编码位数：组索引位数 s、 e 、块内偏移位数 b

写出各类 Cache 的读策略与写策略

掌握 Valgrind 与 Gprof 的使用方法

第 2 章 实验预习

2.1 画出存储器层级结构，标识容量价格速度等指标变化（5 分）



2.2 用 CPUZ 等查看你的计算机 Cache 各参数，写出各级 Cache 的 C S E B s e b（5 分）

1. L1 数据缓存

C = 32 KB, B = 64, b = 6, E = 8, e = 3, S = 64, s = 6

2. L1 指令缓存

C = 32 KB, B = 64, b = 6, E = 8, e = 3, S = 64, s = 6

3. L2 缓存

C = 256KB, B = 64, b = 6, E = 4, e = 2, S = 1024, s = 10

4. L3 缓存

C = 8MB, B = 64, b = 6, E = 16, e = 4, S = 8912, s = 13

2. 3 写出各类 Cache 的读策略与写策略（5 分）

Cache 的读策略：

在高速缓存中查找所需字 w 的副本，如果命中，立即返回字 w 给 CPU，如果不命中，从存储器层级结构中较低层中取出包含字 w 的块，将这个块存储到某个

高速缓存行中（可能会驱逐一个有效的行），然后返回字 w。

Cache 的写策略：

写命中时：

方法 1：直写

立即将 w 的高速缓存块写回到紧接着的低一层中。虽然简单，但是直写的缺点是每次写都会引起总线流量。

方法 2：写回

尽可能地推迟更新，只有当替换算法要驱逐这个更新过的块时，才把它写到紧接着的低一层中。由于局限性，写回能显著地减少总线流量，但是它的缺点是增加了复杂性。高速缓存必须为每个高速缓存行维护一个额外的修改位，表明这个高速缓存块是否被修改过。

写不命中时：

方法 1：写分配

加载相应的低一层中的块到高速缓存中，然后更新这个高速缓存块。写分配试图利用写的空间局部性，但是缺点是每次不命中都会导致一个块从低一层传送到高速缓存。

方法 2：非写分配

避开高速缓存，直接把这个字写到低一层中。直写高速缓存通常是非写分配的。写回高速缓存通常是写分配的。

2.4 写出用 gprof 进行性能分析的方法（5 分）

gprof 是 GNU profile 工具，可以运行于 linux、AIX、Sun 等操作系统进行 C、C++、Pascal、Fortran 程序的性能分析，用于程序的性能优化以及程序瓶颈问题的查找和解决。通过分析应用程序运行时产生的“flat profile”，可以得到每个函数的调用次数，每个函数消耗的处理器时间，也可以得到函数的“调用关系图”，包括函数调用的层次关系，每个函数调用花费了多少时间。使用步骤如下：

（1）用 gcc、g++、xlC 编译程序时，使用-pg 参数，如：g++ -pg -o test.exe test.cpp 编译器会自动在目标代码中插入用于性能测试的代码片断，这些代码在程序运行时采集并记录函数的调用关系和调用次数，并记录函数自身执行时间和被调用函数的执行时间。

（2）执行编译后的可执行程序，如：./test.exe。该步骤运行程序的时间会稍慢于正常编译的可执行程序的运行时间。程序运行结束后，会在程序所在路径下生成一个缺省文件名为 gmon.out 的文件，这个文件就是记录程序运行的性能、调用关系、调用次数等信息的数据文件。

（3）使用 gprof 命令来分析记录程序运行信息的 gmon.out 文件，如：gprof test.exe gmon.out 则可以在显示器上看到函数调用相关的统计、分析信息。上述信息也可以采用 gprof test.exe gmon.out> gprofresult.txt 重定向到文本文件以便于后续分析。

随便找了个程序试了一下：

```
1190200717lh@Graham:~/cbproject/xyz$ gprof main.c gmon.out
gprof: main.c: not in executable format
1190200717lh@Graham:~/cbproject/xyz$ gprof gprofTest gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```

2.5 写出用 Valgrind 进行性能分析的方法（（5 分）

Valgrin 支持很多工具:memcheck，addrcheck，cachegrind，Massif，helgrind 和 Callgrind 等。

基本选项：

-h --help
显示所有选项的帮助，包括内核和选定的工具两者。

--help-debug
和--help 相同，并且还能显示通常只有 Valgrind 的开发人员使用的调试选项。

--version
显示 Valgrind 内核的版本号。工具可以有他们自己的版本号。这是一种保证工具只在它们可以运行的内核上工作的一种设置。这样可以减少在工具和内核之间版本兼容性导致奇怪问题的概率。

-q --quiet
安静的运行，只打印错误信息。在进行回归测试或者有其它的自动化测试机制时会非常有用。

-v --verbose
显示详细信息。在各个方面显示你的程序的额外信息，例如：共享对象加载，使用的重置，执行引擎和工具的进程，异常行为的警告信息。重复这个标记可以增加详细的级别。

-d
调试 Valgrind 自身发出的信息。通常只有 Valgrind 开发人员对此感兴趣。重复这个标记可以产生更详细的输出。如果你希望发送一个 bug 报告，通过-v -v -d -d 生成的输出会使你的报告更加有效。

--tool=<toolname> [default: memcheck]
运行 toolname 指定的 Valgrind，例如，Memcheck, Addrcheck, Cachegrind,等等。

--trace-children=<yes|no> [default: no]
当这个选项打开时，Valgrind 会跟踪到子进程中。这经常会导致困惑，而且通常不是你所期望的，所以默认这个选项是关闭的。

--track-fds=<yes|no> [default: no]
当这个选项打开时，Valgrind 会在退出时打印一个打开文件描述符的列表。每个文件描述符都会打印出一个文件是在哪里打开的栈回溯，和任何与此文件描述符相

关的详细信息比如文件名或 socket 信息。

举个例子：

```
1190200717lh@Graham:~/cbproject/xyz$ valgrind --tool=memcheck --leak-check=full ./valgrindTest
==11355== Memcheck, a memory error detector
==11355== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11355== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==11355== Command: ./valgrindTest
==11355==
1,0x1ffefffd8c
0x1fff0001f3
(nil)
0x1fff000202
0x1fff000222
x = 1190200717
y = 1903008.071700
z = 1190200717-梁浩
The address of x is :0x10c020
The address of y is :0x1ffefffd90
The address of z is :0x10c040
==11355==
==11355== HEAP SUMMARY:
==11355==     in use at exit: 0 bytes in 0 blocks
==11355==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11355==
==11355== All heap blocks were freed -- no leaks are possible
==11355==
==11355== For lists of detected and suppressed errors, rerun with: -s
==11355== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

第 3 章 Cache 模拟与测试

3.1 Cache 模拟器设计

提交 csim.c

程序设计思想：

要做的就是完成 csim.c 中的三个函数：

1）initCache() //为 cache 分配空间，初始化参数

先分配 S 个指针，用这些指针指向 cache 中的每一组，然后申请出每组的内存空间，每组所需的空间大小为 $E*sizeof(cache\_line\_t)$。之后对于每组中每行的参数进行初始化，这里全初始化为 0 了。

```
//cache t cache
//typedef cache_set_t* cache_t
//typedef cache_line_t* cache_set_t
cache = malloc(S*sizeof(cache_set_t));//先分配s个指针，这些指针用来指向每一组
for(int i = 0; i < S ;i++){
    cache[i] = malloc(E*sizeof(cache_line_t));
    //申请出每组的空间，空间大小为行数E*行的大小，让之前申请的每个指针指向每一组
    for(int j = 0 ; j < E ; j++){
        cache[i][j].valid = '0';
        cache[i][j].tag = 0;
        cache[i][j].lru = 0;
    }//对每行中的内容进行初始化
}
```

2）freeCache() // free allocated memory

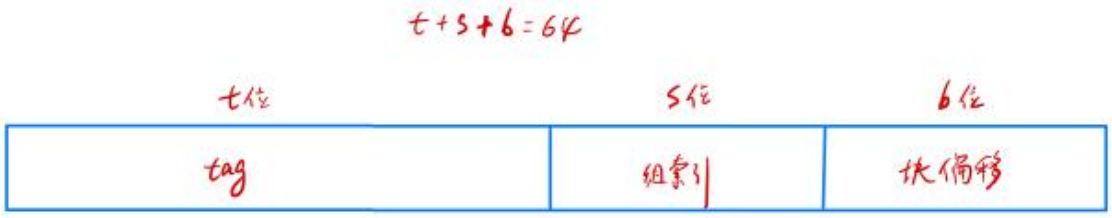
如果要释放 initCache 申请的空间，可以先把每组占据的空间释放，再将指向每组的指针释放。

```
for (int i = 0; i < S; i++) {
    free(cache[i]);//这里是将每组占据的空间释放了
}
//还要释放掉最开始申请的指向每组的那s个指针的空间
free(cache);
```

3）accessData(mem_addr_t addr)

函数会传入一个地址 addr，可以先对 addr 进行移位操作来获取它的组索引和 tag。由于只是要判断在 cache 是否命中（如果不命中直接加入缓存中），所以可以不考虑块偏移。

addr 的结构如图所示：



要想获取 tag 的值，只需要将 addr 逻辑右移 $(b+s)$ 个单位。要想获取组索引，先将 addr 左移 t 个单位，再右移 $(b+t)$ 个单位。

```
int t = 64 - s - b;//这里的t位表示标记，64位地址
unsigned long long int zu_index = (addr << t) >> (b + t);//获取组索引
unsigned long long int tag = addr >> (s + b);//获取标记
```

获取了标记值和组索引之后就可以判断是否命中了。

如果命中要满足两个条件：标记值相等，有效位为 1

cache[zu_index][i].tag == tag && cache[zu_index][i].valid == '1'

命中的话，命中总次数加 1，时间戳加 1，命中的那一行的 lru 等于现在的时间戳。（每行的 lru 参数用来记录它上一次被操作是在什么时候，如果 lru 小，说明上一次操作它的时间很远）

```
hit_count++;
lru_counter++;
cache[zu_index][i].lru = lru_counter;
return;
```

如果不命中，就要将 addr 存入 cache 之中，分两种情况：

i. 对应组里有空行

如果有空行（利用 valid 有效位是否为 0 来判断是否为空），直接将该行的 tag 值改为 addr 对应的 tag 值，valid 位改为'1'， 时间戳加 1，当前行的 lru 等于现在的时间戳，miss 的总次数加 1。

```
for (int i = 0; i < E; i++) {
    if (cache[zu_index][i].valid == '0') {
        cache[zu_index][i].valid = '1';
        cache[zu_index][i].tag = tag;
        lru_counter++;
        cache[zu_index][i].lru = lru_counter;
        miss_count++;
        return;
    }
}
```

ii. 没有空行，需要替换，替换策略为：替换当前组里 lru 最小的那个，说明它已经很久没有被操作过了。

遍历当前组，找到 lru 最小的行，将行参数替换为 addr 的参数。

```
int min_lru_index = 0;
unsigned long long int min_lru = cache[zu_index][0].lru;
for (int i = 1; i < E; i++) {
    if (cache[zu_index][i].lru < min_lru) {
        min_lru = cache[zu_index][i].lru;
        min_lru_index = i;
    }//如果遇到更小的lru
}
//到这里就找到了用最小lru的行的下标，为min_lru_index，只要替换它即可
cache[zu_index][min_lru_index].tag = tag;
lru_counter++;
cache[zu_index][min_lru_index].lru = lru_counter;
return;
```

测试用例 1 的输出截图（5 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
```

测试用例 2 的输出截图（5 分）：

```
190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
its:4 misses:5 evictions:2
190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 4 -E 2 -b 4 -t traces/yi.trace
its:4 misses:5 evictions:2
```

测试用例 3 的输出截图（5 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
its:2 misses:3 evictions:1
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 2 -E 1 -b 4 -t traces/dave.trace
its:2 misses:3 evictions:1
```

测试用例 4 的输出截图（5 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
hits:167 misses:71 evictions:67
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 2 -E 1 -b 3 -t traces/trans.trace
hits:167 misses:71 evictions:67
```

测试用例 5 的输出截图（5 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
hits:201 misses:37 evictions:29
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 2 -E 2 -b 3 -t traces/trans.trace
hits:201 misses:37 evictions:29
```

测试用例 6 的输出截图（5 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
hits:212 misses:26 evictions:10
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 2 -E 4 -b 3 -t traces/trans.trace
hits:212 misses:26 evictions:10
```

测试用例 7 的输出截图（5 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
hits:231 misses:7 evictions:0
1190200717lh@Graham:~/lab6/cachelab-handout$
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 5 -E 1 -b 5 -t traces/trans.trace
hits:231 misses:7 evictions:0
```

测试用例 8 的输出截图（10 分）：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
1190200717lh@Graham:~/lab6/cachelab-handout$ ./csim-ref -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
```

注：每个用例的每一指标 5 分（最后一个用例 10）——与参考 csim-ref 模拟器输出指标相同则判为正确

总的如下：

```
1190200717lh@Graham:~/lab6/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
```

3.2 矩阵转置设计

提交 trans.c

程序设计思想：

32 × 32 (M = 32, N = 32):

如果按照最普通的方式进行转置，遍历每行的时候遍历每列，

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

会发现 misses 数过大

```
Summary for official submission (func 0): correctness=1 misses=1183

TEST_TRANS_RESULTS=1:1183
```

根据 PPT，可以利用分块矩阵的思想来优化代码，减少 misses 数。

可知 cache 的各个参数为: s = 5, b = 5, E = 1

cache 的结构可以画成这样：

0
1
2
3
.....
28
29
30
31

该 cache 共有 32 组，每组一行，一行的块大小为 32 个字节，即 8 个 int。

那么利用矩阵分块的思想，可以尝试将矩阵分为 16 个 8*8 大小的矩阵。

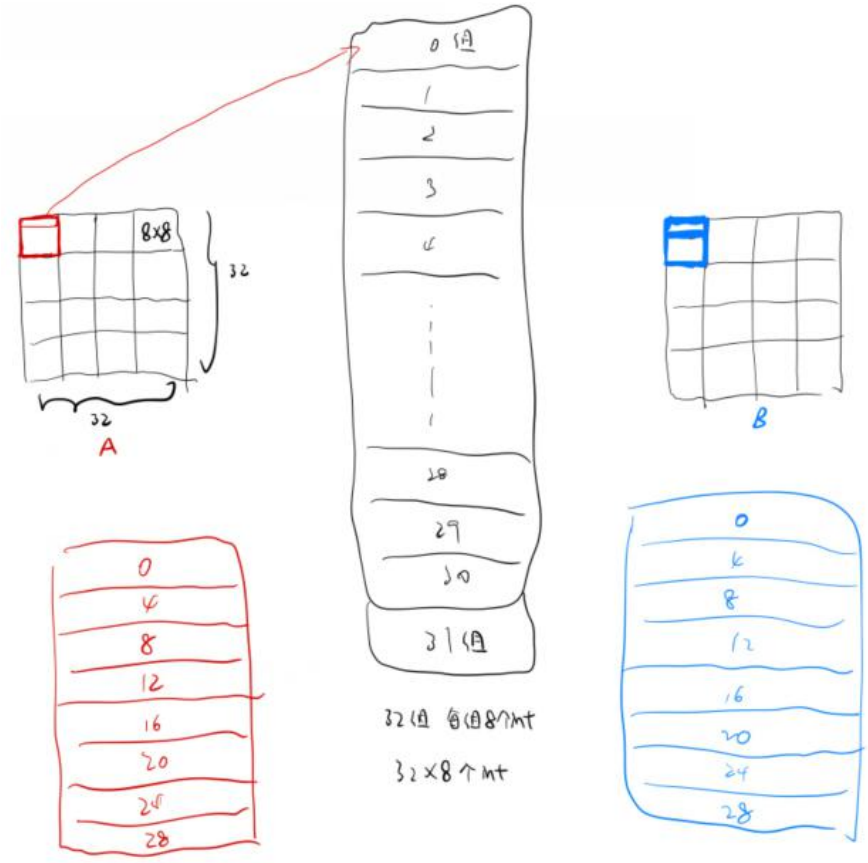
代码如下：

```
for (int i = 0; i < M; i += 8) {
    for (int j = 0; j < N; j += 8) {
        for (int m = i; m < i + 8; m++) {
            for (int n = j; n < j + 8; n++) {
                B[n][m] = A[m][n];
            }
        }
    }
}
```

跑一遍看看效果，

```
Summary for official submission (func 0): correctness=1 misses=343

TEST_TRANS_RESULTS=1:343
```

misses 数下降了很多，但还是没有达到最优，接下来仔细分析转置的过程。



首先看 A 矩阵中最左上角的 8*8 矩阵，通过分析可以知道它如果放到 cache 里，每行的 8 个 int 对应的组如上图所示（这里不妨假设第一行对应 cache 的第 0 组），这个红色矩阵转置后也对应 B 中最左上角的 8*8 矩阵（标为蓝颜色），这个蓝颜色的 8*8 矩阵每行中的 8 个 int 如果要放入 cache 中对应的组由上图所示。观察可以发现，红色矩阵和蓝色矩阵如果要放到 cache 里放入的组是相同的。这就会发生相

互替换，导致冲突不命中。

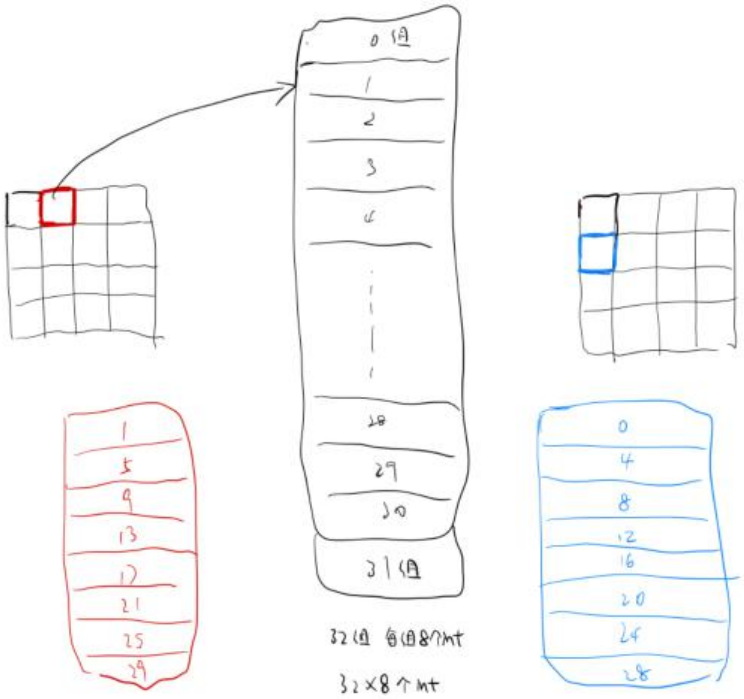
举个例子：

最开始访问 A[0][0]时，cache 里没东西，会发生冷不命中，然后将 A[0][0]-A[0][7]放入 cache 的第 0 组。

访问 B[0][0]时，它对应的组索引也是在第 0 组，此时第 0 组放的是 A[0][0]-A[0][7]，显然 tag 是不相等的，会发生冲突不命中，会将 B[0][0]-B[0][7]放入第 0 组中。

访问 A[0][1]时，它对应的组索引也是在第 0 组，此时第 0 组放的是 B[0][0]-B[0][7]，显然 tag 是不相等的，会发生冲突不命中，会将 A[0][0]-A[0][7]放入第 0 组中。

由此，可以发现因为红色矩阵和蓝色矩阵如果要放到 cache 里放入的组是相同的，因此会发生这种相互替换的情况。对于这种的分块方法，对角线的块都会出现这种情况。非对角线的块由于放入的组不相同不会出现这种情况（如下图所示）。



那么只要避免这种相互替换的情况就可以实现优化。

相互替换的情形坏就坏在一开始在把 A[0][0]-A[0][7]放入 cache 的第 0 组时没有充分利用好这 8 个数就已经因为要访问 B[0][0]而被 B[0][0]-B[0][7]给替换了，因此可以先用好 A 的这 8 个数再访问 B。如下图所示：

```
int a0, a1, a2, a3, a4, a5, a6, a7;

if (M == 32 && N == 32) {
    for (int i = 0; i < M; i += 8) {
        for (int j = 0; j < N; j += 8) {
            for (int m = i; m < i + 8; m++) {
                if (i == j) {
                    a0 = A[m][j];
                    a1 = A[m][j + 1];
                    a2 = A[m][j + 2];
                    a3 = A[m][j + 3];
                    a4 = A[m][j + 4];
                    a5 = A[m][j + 5];
                    a6 = A[m][j + 6];
                    a7 = A[m][j + 7];
                    B[j][m] = a0;
                    B[j + 1][m] = a1;
                    B[j + 2][m] = a2;
                    B[j + 3][m] = a3;
                    B[j + 4][m] = a4;
                    B[j + 5][m] = a5;
                    B[j + 6][m] = a6;
                    B[j + 7][m] = a7;
                }
                else {
                    for (int n = j; n < j + 8; n++) {
                        B[n][m] = A[m][n];
                    }
                }

            }
        }
    }
}
```

测试后效果：

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

TEST_TRANS_RESULTS=1:287
```

效果好了很多。

64 × 64 (M = 64, N = 64):

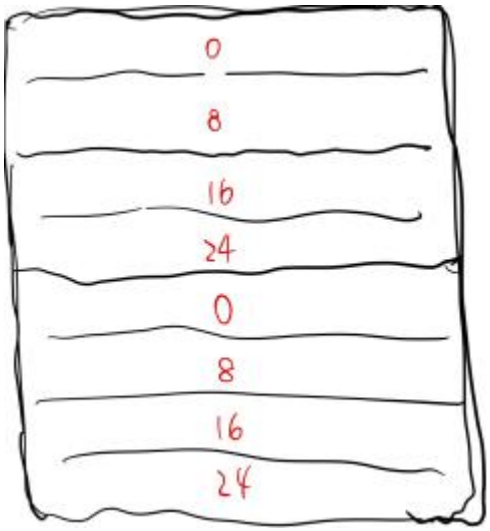
如果按照 32*32 的方法来处理，分成 8*8 的小块，每次用 8 个变量保存 A 中当前行的内容，如下图所示：

```
if( M == 64 && N == 64){
    for (i = 0; i < M; i+=8) {
            for (j = 0; j < N; j+=8) {
                    for (m = i; m < i + 8; m++) {

                a0 = A[m][j];
                a1 = A[m][j + 1];
                a2 = A[m][j + 2];
                a3 = A[m][j + 3];
                a4 = A[m][j + 4];
                a5 = A[m][j + 5];
                a6 = A[m][j + 6];
                a7 = A[m][j + 7];

                B[j][m] = a0;
                B[j + 1][m] = a1;
                B[j + 2][m] = a2;
                B[j + 3][m] = a3;
                B[j + 4][m] = a4;
                B[j + 5][m] = a5;
                B[j + 6][m] = a6;
                B[j + 7][m] = a7;

                    }
            }
    }
}
```

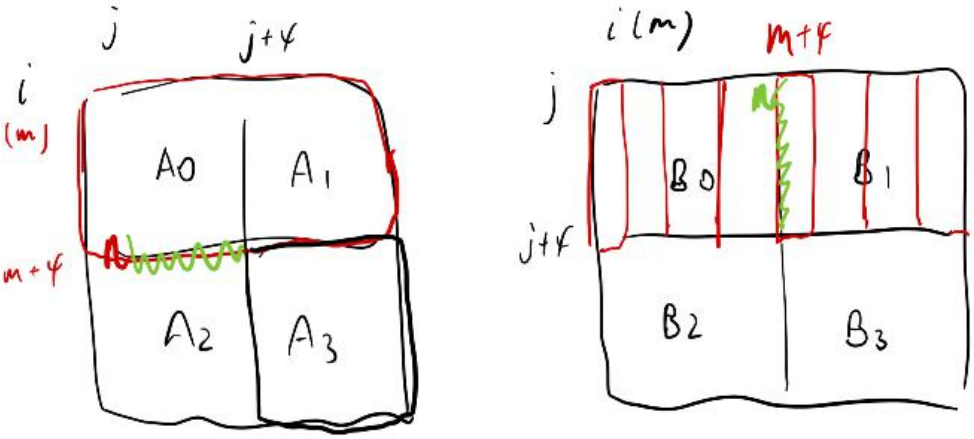
运行后的结果为下图所示，和要求的情况相差挺多。

```
Summary for official submission (func 0): correctness=1 misses=4611

TEST_TRANS_RESULTS=1:4611
```

接下来分析一下为什么 misses 数会这么多，



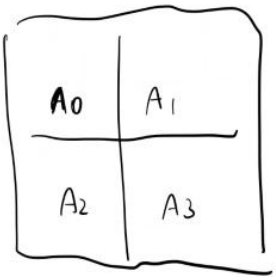
上图是一个 8*8 的矩阵，由于现在矩阵大小是 64*64，所以在 8*8 的矩阵中也出现放入 cache 后会出现放在同一组的情况。因此，为了避免这种情况的出现，可以考虑将这个 8*8 的矩阵再分成 4 个 4*4 大小的块，如下图所示：

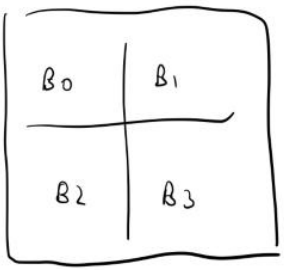


问题就转化成了如何利用这 4 个 4*4 的小块实现 8*8 大小的转置。

下面展现大致的操作过程：

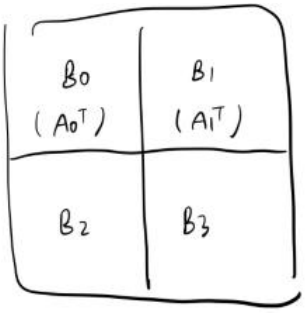
初始状态如图所示：





先将 A_0 和 A_1 转置后的 A_0^T 和 A_1^T 放入 B_0 和 B_1 的位置

```
for (m = i; m < i+4; m++) {
    B[j][m] = A[m][j];
    B[j+1][m] = A[m][j+1];
    B[j+2][m] = A[m][j+2];
    B[j+3][m] = A[m][j+3];
    B[j][m+4] = A[m][j+4];
    B[j+1][m+4] = A[m][j+5];
    B[j+2][m+4] = A[m][j+6];
    B[j+3][m+4] = A[m][j+7];

}
```

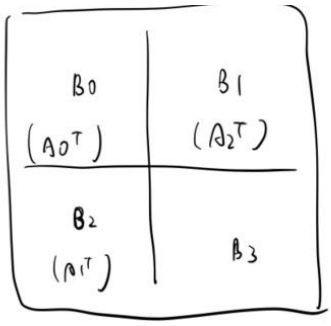


这一步的好处在于在逐行访问 A_0 元素时，由于 A_0 和 A_1 会被同时放入 cache 中就会减少 misses 量， B_0 和 B_1 也是如此。

到此在 cache 中的有 A_0 和 A_1， B_0 和 B_1。

然后将按列取出 A_2 的元素（此时由于 A_0 和 A_2 争缓存， A_0 和 A_1 已经不在缓存中了），按行取出 B_1 的元素，接下来注意要先将 A_2 的列放入 B_1 的行，再将 B_1 的行放入 B_2 中。这样的好处在于：如果先将 B_1 的行放入 B_2 中， B_2 会和 B_1 争缓存，此时缓存中存的是 B_2 和 B_3，再想将 A_2 的列放入 B_1 的行中时， A_2 不在缓存中，原本缓存中 A_0 和 A_1 会被 A_2 替换，同时 B_1 此时已经不再缓存中了，又要重新替换一遍。但是如果先将 A_2 的列放入 B_1 的行中时，A_2 依旧会替换 A_0 和 A_1 的位置，但是此时 B_1 在缓存中，因此可以直接赋值，再将 B_1 的行放入 B_2 中时， B_1 已经在， B_2 只需

要替换 B_1 即可。相比之下，争缓存的次数少了 1 次。



到此在 cache 中的有 A_2 和 A_3， B_2 和 B_3 。

此时想要将 A_3^T 放入 B_3 中时，由于它们都在缓存中，就会变得非常简单。

运行后结果为：

```
Summary for official submission (func 0): correctness=1 misses=1924

TEST_TRANS_RESULTS=1:1924
```

满足要求。

61 × 67 (M = 61, N = 67)：

对于 61*67 的矩阵，可以考虑分成 16*4 的小块，

```
if (M == 61 && N == 67) {
    for (i = 0; i < 67; i += 16) {
        for (j = 0; j < 61; j += 4) {
            for (m = i; m < i + 16 && m < N; m++) {
                for (n = j; n < j + 4 && j < M; n++) {
                    B[n][m] = A[m][n];
                }
            }
        }
    }
}
```

运行后结果为 1990，符合要求，如下图所示：

```
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:6591, misses:1990, evictions:1958

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:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1990

TEST_TRANS_RESULTS=1:1990
```

32×32（10 分）：运行结果截图

```
1190200717lh@Graham:~/lab6/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

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

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

TEST_TRANS_RESULTS=1:287
```

64×64（10 分）：运行结果截图

```
1190200717lh@Graham:~/lab6/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:8962, misses:1283, evictions:1251

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:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1283

TEST_TRANS_RESULTS=1:1283
```

61×67（20 分）：运行结果截图

```
1190200717lh@Graham:~/lab6/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:6591, misses:1990, evictions:1958

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:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1990

TEST_TRANS_RESULTS=1:1990
```

driver.py 的运行跑分如下：

```
1190200717lh@Graham:~/lab6/cachelab-handout$ python2 driver.py
Part A: Testing cache simulator
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        1283
Trans perf 61x67          10.0        10        1990
          Total points    53.0        53
```

第 4 章 总结

4.1 请总结本次实验的收获

进一步理解了 cache 的基本知识；
进一步了解了 cache 命中和不命中情况下的处理方法；
尝试了用矩阵分块的思想降低矩阵转置 misses 数的方法。

4.2 请给出对本次实验内容的建议

无

注：本章为酌情加分项。

参考文献

[1] 深入理解计算机系统