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　计算机类

学　　号　1190200910

班　　级　1903012

学　　生　严幸

指 导 教 师　史先俊

实 验 地 点　G709

实 验 日 期　2021-05-26

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

- X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

- Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

- Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

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 Cache 的基本结构与参数：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 Cache 的各种读策略与写策略
- 掌握 Valgrind、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）

注：此图摘自《深入理解计算机系统》P421 页

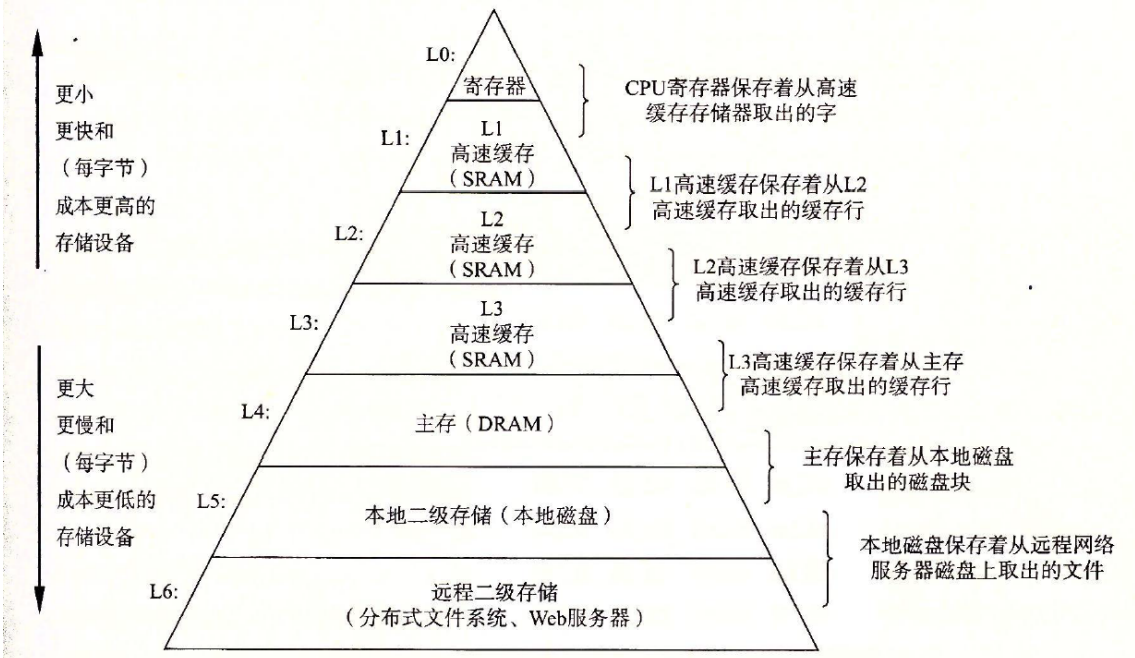


图 6-21 存储器层次结构

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）

一级数据 Cache：C=48KB×4, E=12, B=64, S=256, s=8, e=log2(12), b=6

一级指令 Cache：C=32KB×4, E=8, B=64, S=256, s=8, e=3, b=6

二级 Cache：C=1280KB×4, E=20, B=64, S=4096, s=12, e=log2(20), b=6

三级 Cache：C=8MB, E=8, B=64, S=16384, s=14, e=3, b=6

2.3 写出各类 Cache 的读策略与写策略（5 分）

1) 读策略：

读取第 k 级 cache，

命中时，直接从 Cache 中取数送到 CPU 寄存器或送到上一级 cache。

不命中时，从第 k+1 级 cache 中或从内存中取数据存储在第 k 级 cache 中。如果第 k 级 cache 的对应组有空闲区域(即有效位为 0)，则直接在空闲区域写入；如果没有空闲区域，则淘汰某一路的数据之后再写入，一般采用 LRU 算法选择要淘汰的路。

2) 写策略

写策略在**命中**时，有两种策略，一是“写回”，二是“直写”。

写回法：命中第 k 级 cache 时，只写第 k 级 cache，等到第 k 级 cache 的这一行被淘汰时，才将数据写回第 k+1 级 cache。

直写法：命中第 k 级 cache 时，将每一级的 cache 中对应的数据都进行重写，

内存中的数据也要重写，这样在淘汰那一行的数据时就不用再写了。

写策略在**不命中**时，也有两种策略，一是“写分配”，二是“非写分配”。

写分配：在不命中时，加载低一层 cache 中的块到本层 cache 中，然后更新这个 cache 块。

非写分配：避开本层 cache，直接把字写到低一层的 cache 中。

2.4 写出用 gprof 进行性能分析的方法（5 分）

注：本部分报告参考了博客[7]。

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。

常用的 gprof 命令选项：

-b 不再输出统计图表中每个字段的详细描述。

-p 只输出函数的调用图（Call graph 的那部分信息）。

-q 只输出函数的时间消耗列表。

-e Name 不再输出函数 Name 及其子函数的调用图（除非它们有未被限制的其它父函数）。可以给定多个 -e 标志。一个 -e 标志只能指定一个函数。

-E Name 不再输出函数 Name 及其子函数的调用图，此标志类似于 -e 标志，但它在总时间和百分比时间的计算中排除了由函数 Name 及其子函数所用的时间。

-f Name 输出函数 Name 及其子函数的调用图。可以指定多个 -f 标志。一个 -f 标志只能指定一个函数。

-F Name 输出函数 Name 及其子函数的调用图，它类似于 -f 标志，但它在总时间和百分比时间计算中仅使用所打印的例程的时间。可以指定多个 -F 标志。一个 -F 标志只能指定一个函数。-F 标志覆盖 -E 标志。

-z 显示使用次数为零的例程（按照调用计数和累积时间计算）。

使用流程：

1. 在编译和链接时 加上-pg 选项。一般我们可以加在 makefile 中。

2. 执行编译的二进制程序。执行参数和方式同以前。

3. 在程序运行目录下 生成 gmon.out 文件。如果原来有 gmon.out 文件，将会被重写。

4. 结束进程。这时 gmon.out 会再次被刷新。

5. 用 gprof 工具分析 gmon.out 文件。

2.5 写出用 Valgrind 进行性能分析的方法（5 分）

Valgrind 是运行在 Linux 上一套基于仿真技术的程序调试和分析工具，它包含一个内核——一个软件合成的 CPU，和一系列的小工具，每个工具都可以完成一项任务——调试，分析，或测试等。

用法: valgrind [options] prog-and-args

[options]: 常用选项，适用于所有 Valgrind 工具

-tool=<name> 最常用的选项。运行 valgrind 中名为 toolname 的工具。默认 memcheck。

memcheck ------> 这是 valgrind 应用最广泛的工具,一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。

callgrind ------> 它主要用来检查程序中函数调用过程中出现的问题。

cachegrind ------> 它主要用来检查程序中缓存使用出现的问题。

helgrind ------> 它主要用来检查多线程程序中出现的竞争问题。

massif ------> 它主要用来检查程序中堆栈使用中出现的问题。

extension ------> 可以利用 core 提供的功能，自己编写特定的内存调试工具

-h -help 显示帮助信息。

-version 显示 valgrind 内核的版本，每个工具都有各自的版本。

-q -quiet 安静地运行，只打印错误信息。

-v -verbose 更详细的信息, 增加错误数统计。

-trace-children=no|yes 跟踪子线程? [no]

-track-fds=no|yes 跟踪打开的文件描述？ [no]

-time-stamp=no|yes 增加时间戳到 LOG 信息? [no]

-log-fd=<number> 输出 LOG 到描述符文件 [2=stderr]

-log-file=<file> 将输出的信息写入到 filename.PID 的文件里，PID 是运行程序的进行 ID

-log-file-exactly=<file> 输出 LOG 信息到 file

-log-file-qualifier=<VAR> 取得环境变量的值来做为输出信息的文件名。[none]

-log-socket=ipaddr:port 输出 LOG 到 socket ，ipaddr:port

LOG 信息输出

-xml=yes 将信息以 xml 格式输出，只有 memcheck 可用

-num-callers=<number> show <number> callers in stack traces [12]

-error-limit=no|yes 如果太多错误，则停止显示新错误? [yes]

-error-exitcode=<number> 如果发现错误则返回错误代码 [0=disable]

-db-attach=no|yes 当出现错误，valgrind 会自动启动调试器 gdb。[no]

-db-command=<command> 启动调试器的命令行选项[gdb -nw %f %p]

适用于 Memcheck 工具的相关选项：

-leak-check=no|summary|full 要求对 leak 给出详细信息? [summary]

-leak-resolution=low|med|high how much bt merging in leak check [low]

-show-reachable=no|yes show reachable blocks in leak check? [no]

最常用的命令格式：

valgrind --tool=memcheck --leak-check=full ./test

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：

```
/*
 * initCache - Allocate memory, write 0's for valid and tag and LRU
 * also computes the set_index_mask
 */
void initCache()
{
 //todo...

}


/*
 * freeCache - free allocated memory
 */
void freeCache()
{
 //todo...

}


/*
 * accessData - Access data at memory address addr.
 *   If it is already in cache, increast hit_count
 *   If it is not in cache, bring it in cache, increase miss count.
 *   Also increase eviction_count if a line is evicted.
 */
void accessData(mem_addr_t addr)
{
 //todo...

}
```

```
260
261     /* Compute S, E and B from command line args */
262     S = myPow(2, s);
263     B = myPow(2, b);
264     // printf("%d %d %d", S, B, E);
265     // return 0;
266     /* Initialize cache */
267     initCache();
268
269 #ifdef DEBUG_ON
270     printf("DEBUG: S:%u E:%u B:%u trace:%s\n", S, E, B, trace_file);
271     printf("DEBUG: set_index_mask: %llu\n", set_index_mask);
272 #endif
273
274     replayTrace(trace_file);
275
276     /* Free allocated memory */
277     freeCache();
278
279     /* Output the hit and miss statistics for the autograder */
280     printSummary(hit_count, miss_count, eviction_count);
281     return 0;
```

程序主要需要我们设计一开始 S, E, B 的计算、缓存初始化部分、缓存空间回收部分、访问缓存数据部分。

对于一开始 S, E, B 的计算，只需要计算 2 的 s 次方得到 S，2 的 b 次方得到 B 即可。E 不需要计算。

```
55  void initCache()
56  {
57      cache = (cache_set_t *)malloc(sizeof(cache_set_t) * S);
58      for (int i = 0; i < S; i++)
59      {
60          cache[i] = (cache_line_t *)malloc(sizeof(cache_line_t) * E);
61          for (int j = 0; j < E; j++){
62              cache[i][j].lru = 0;
63              cache[i][j].tag = 0;
64              cache[i][j].valid = 0;
65          }
66      }
67  }
```

在缓存初始化时，给 cache 分配 S 个组空间，每个组分配 E 条路的空间，并把每一条路的 lru、tag、valid 均置为 0。

```
73  void freeCache()
74  {
75      for (int i = 0; i < S; i++)
76      {
77          free(cache[i]);
78      }
79      free(cache);
80  }
```

在释放缓存时，释放每一路的空间，最后再释放每一组的空间即可。

访问某个地址时，首先要根据地址把组号、标记位给拿出来，放到 addr_set_num 和 addr_tag 变量中。

```
91  void accessData(mem_addr_t addr)
92  {
93      //取出addr最右边的b位，即为块内偏移
94      int t = ADDRESS_LENGTH - s - b; //得到标记位的长度
95      mem_addr_t tmp = addr;
96      mem_addr_t addr_tag = tmp >> (b + s);
97      int addr_set_num = ((tmp << t) >> t) >> b; //得到组号
98      int hasEmpty = 0;
99      int isHit = 0;
100     //该组是cache[addr_set_num]
```

然后依次检查这一组的每一路的 tag 和有效位，看是否命中。如果命中则让 hit_count 加一。代码中的 lru_counter 是为了 LRU 算法淘汰时使用，在后面介绍 LRU 的具体实现。在遍历每一路时，同时记录下是否有空闲的路，为不命中做准备。

```
//该组是cache[addr_set_num]
for (int i = 0; i < E; i++)
{                                                                    //依次检查该组的每一路
    if (cache[addr_set_num][i].tag == addr_tag && cache[addr_set_num][i].valid == 1) //命中
    {
        hit_count++;
        lru_counter++;
        cache[addr_set_num][i].lru = lru_counter;
        isHit = 1;
        break;
    }
    if (cache[addr_set_num][i].valid == 0)
        hasEmpty = 1;
}
```

如果没有命中，则要看是否有空位，有空位的话直接在空位加载这个地址的

数据，把 tag 和有效位分别赋值。

```
//没有命中
if (!isHit)
{
    miss_count++;
    if (hasEmpty)
    { //如果有空位
        for (int i = 0; i < E; i++)
        {
            if (cache[addr_set_num][i].valid == 0)
            { //找到空位
                lru_counter++;
                cache[addr_set_num][i].valid = 1;
                cache[addr_set_num][i].lru = lru_counter;
                cache[addr_set_num][i].tag = addr_tag;
                break;
            }
        }
    }
```

如果没有命中，则采用 LRU 算法淘汰，然后在淘汰的位置加载数据。

```
else //如果没有空位，则需要淘汰，将lru_counter最小的淘汰
{
    eviction_count++;
    unsigned long long minLru = cache[addr_set_num][0].lru;
    int minLruIdx = 0;
    for (int i = 1; i < E; i++){
        if (cache[addr_set_num][i].lru < minLru){
            minLru = cache[addr_set_num][i].lru;
            minLruIdx = i;
        }
    }
    lru_counter++;
    cache[addr_set_num][minLruIdx].lru = lru_counter;
    cache[addr_set_num][minLruIdx].tag = addr_tag;
    cache[addr_set_num][minLruIdx].valid = 1;
}
```

下面阐述 LRU 算法的实现：

使用一个全局的 lru_counter 变量记录访问次数，这个变量随时间的推移只会增加不会减少，可以理解为一个离散化的时间戳。然后每次访问到某个路时，将这一路的结构体中 lru 变量赋为当时的 lru_counter，就相当于记录了这个块的最后一次访问时间。要淘汰时，选出一组中 lru 最小的一路，就相当于选出了“最后一次访问时间最早”的一路，然后淘汰这一路即可。

测试用例 1 的输出截图（5 分）：

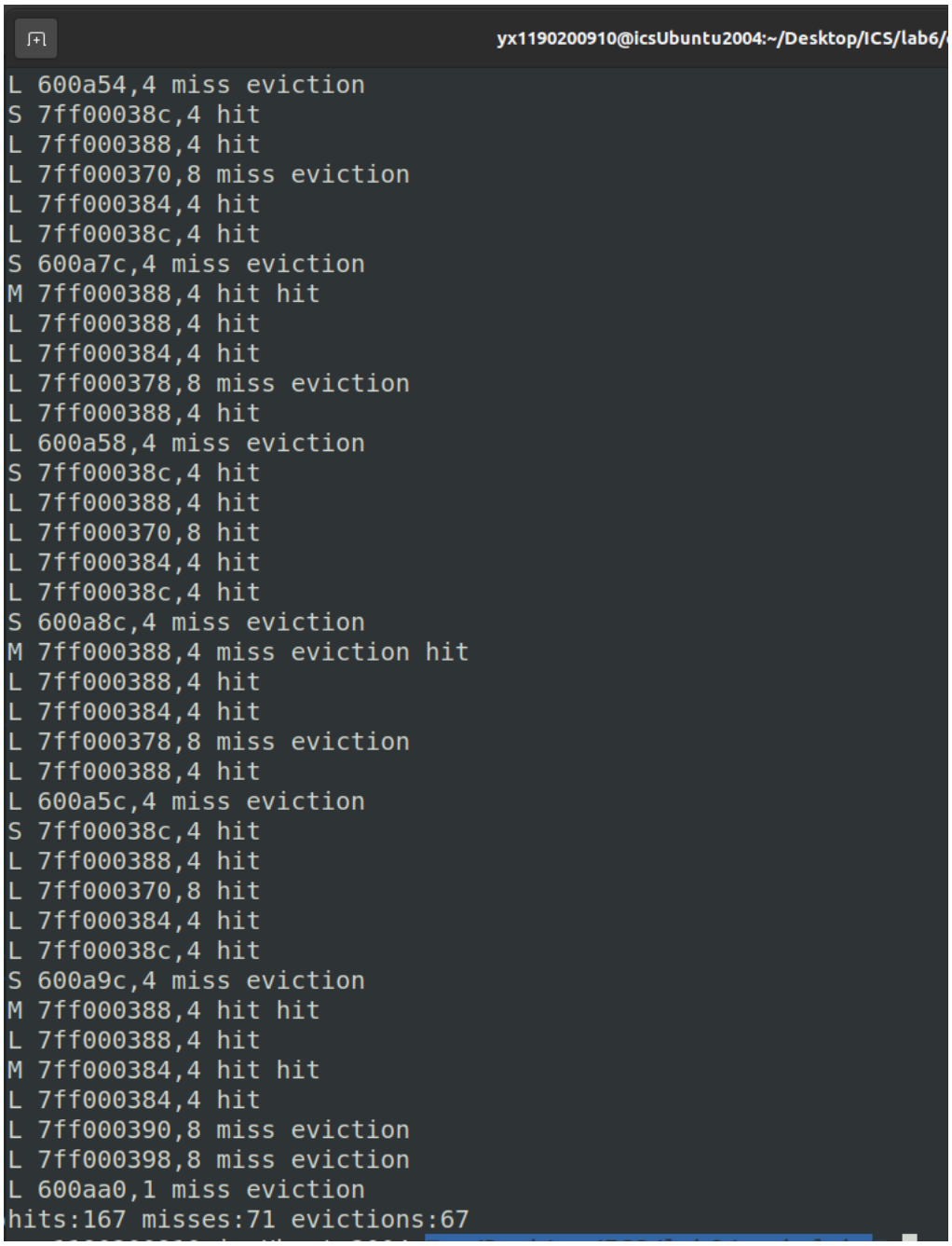
测试用例 2 的输出截图（5 分）：

```
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./csim -v -s 4 -E 2 -b 4 -t traces/yi.tr
ace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:2
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./csim-ref -v -s 4 -E 2 -b 4 -t traces/y
i.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:2
```

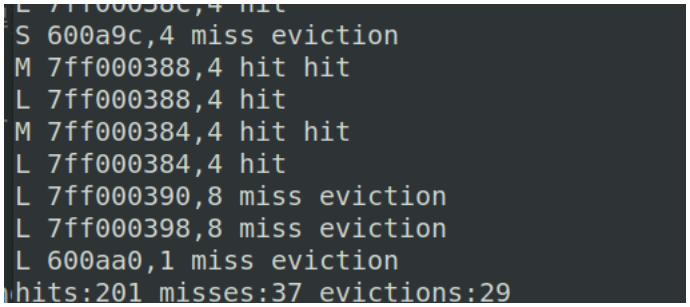
测试用例 3 的输出截图（5 分）：

```
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./csim -v -s 2 -E 1 -b 4 -t traces/dave.
trace
L 10,4 miss
S 18,4 hit
L 20,4 miss
S 28,4 hit
S 50,4 miss eviction
hits:2 misses:3 evictions:1
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./csim-ref -v -s 2 -E 1 -b 4 -t traces/d
ave.trace
L 10,4 miss
S 18,4 hit
L 20,4 miss
S 28,4 hit
S 50,4 miss eviction
hits:2 misses:3 evictions:1
```

测试用例 4 的输出截图（5 分）：



测试用例 5 的输出截图（5 分）：



测试用例 6 的输出截图（5 分）：

```
L 7ff000388,4 hit
L 7ff000370,8 hit
L 7ff000384,4 hit
L 7ff00038c,4 hit
S 600a9c,4 hit
M 7ff000388,4 hit hit
L 7ff000388,4 hit
M 7ff000384,4 hit hit
L 7ff000384,4 hit
L 7ff000390,8 miss eviction
L 7ff000398,8 miss eviction
L 600aa0,1 miss eviction
hits:212 misses:26 evictions:10
```

测试用例 7 的输出截图（5 分）：

```
M 7ff000384,4 hit hit
L 7ff000384,4 hit
L 7ff000390,8 hit
L 7ff000398,8 hit
L 600aa0,1 hit
hits:231 misses:7 evictions:0
```

测试用例 8 的输出截图（10 分）：

```
M 7fefe0594,4 hit hit
L 7fefe0594,4 hit
L 7fefe058c,4 hit
M 7fefe0590,4 hit hit
L 7fefe0590,4 hit
L 7fefe0588,4 hit
L 7fefe05a0,8 miss eviction
L 7fefe05a8,8 hit
S 602265,1 miss eviction
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

测试运行总体截图：

```
 yx1190200910@icsUbuntu2004 ☰ ~/Desktop/ICS/lab6/cachelab ☰ ./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
 yx1190200910@icsUbuntu2004 ☰ ~/Desktop/ICS/lab6/cachelab ☰
```

3.2 矩阵转置设计

提交 trans.c

程序设计思想：

程序所用 cache 的规格是 s=5,E=1,b=5，也就是 S=32，有 32 组；E=1，每组有 1 路；B=32，每一路可以存放 32 字节，也就 8 个 int 型的数。Cache 的大小可以存放 256 个 int 型数。

对于 32×32 的矩阵，很容易想到将矩阵分块成 8*8 的矩阵，可以将 AB 两个 8*8 矩阵全部装入 cache，然后始终能够命中。

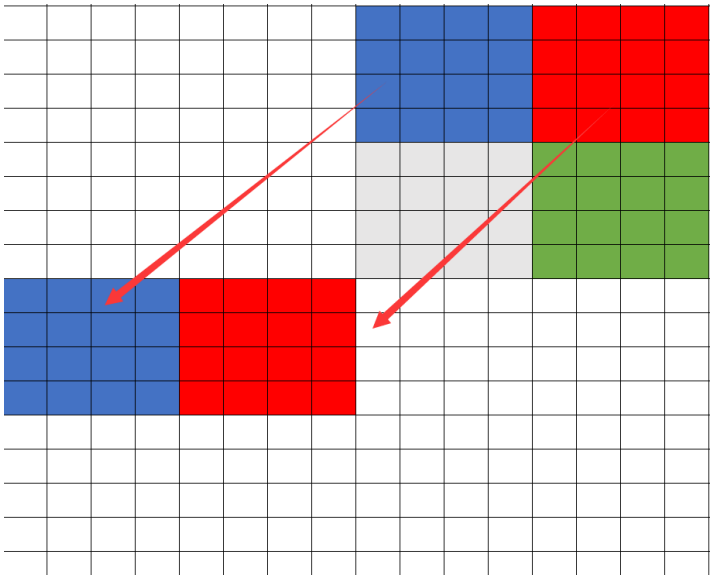
```
if (M == 32 && N == 32)
{
    for (int i = 0; i < M; i += 8)
    {
        for (int j = 0; j < N; j += 8)
        {
            for(int ii = i; ii < i+8; ii++){
                for(int jj = j; jj < j+8; jj++){
                    B[ii][jj] = A[jj][ii];
                }
            }
        }
    }
```

这段代码可达到 343 的 Miss。还不能满足题目 300 以内的 miss 要求。推测可能是因为交替访问 AB 中的元素导致了连续的 AB 互相淘汰，发生了冲突不命中。我们将代码做循环展开，使用临时变量保存 A 中的连续 8 个数，再将它们一一赋给 B，即可达到要求：

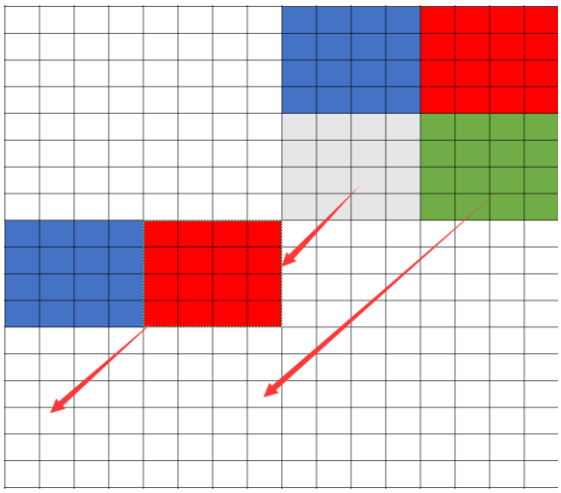
```
if (M == 32 && N == 32)
{
    for (i = 0; i < M; i += 8)
    {
        for (j = 0; j < N; j++)
        {
            t0 = A[j][i]; t1 = A[j][i+1];
            t2 = A[j][i+2]; t3 = A[j][i+3];
            t4 = A[j][i+4]; t5 = A[j][i+5];
            t6 = A[j][i+6]; t7 = A[j][i+7];

            B[i][j] = t0; B[i+1][j] = t1;
            B[i+2][j] = t2; B[i+3][j] = t3;
            B[i+4][j] = t4; B[i+5][j] = t5;
            B[i+6][j] = t6; B[i+7][j] = t7;
        }
    }
}
```

对于 64*64 的情况，仍使用 8*8 分块，会达到 4611 的 miss，无法满题意，推测原因可能是，64 正好等于 8 的平方，元素排列“过于整齐”，发生了**冲突不命中**现象，导致 miss 数较大。由于一次性可以装入 4 行矩阵元素，改用 4*4 分块后可以达到 1651 的 miss 数量，但距离题目要求的 1300 还是差了点。推测原因，可能是每一次装载了 8 个 int 型的数，但是只用到了其中 4 个，另外 4 个装载没有用到，就被下一个同组元素驱逐了，这就带来了额外开销。因此，还是应该基于 8*8 分块的基础上进行改进。参考博客[8]后，我使用了如下方法，将 8*8 分块与 4*4 分块相结合。

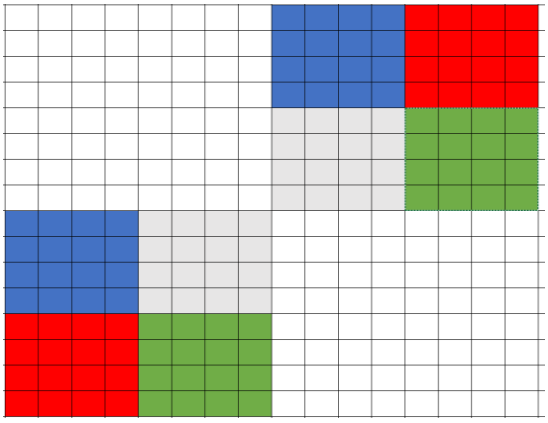
方法总体是在一个 8*8 分块中将 4*4 分块转置用另一种方法实现。



对于上图中的矩阵，先将蓝色和红色区域复制到左下方的相同位置。



然后把复制后的红色区域再往左下复制，灰色区域和绿色区域也复制到对应的转置后位置。就能得到下图的转置后结果。



编写的代码如下：

```
else if(M == 64 && N == 64){
    for (i = 0; i < M; i += 8)
    {
        for (j = 0; j < N; j += 8)
        {
            for (k = i; k < i + 4; k++)
            {
                t1 = A[k][j]; t2 = A[k][j + 1];
                t3 = A[k][j + 2]; t4 = A[k][j + 3];
                t5 = A[k][j + 4]; t6 = A[k][j + 5];
                t7 = A[k][j + 6]; t8 = A[k][j + 7];

                B[j][k] = t1; B[j + 1][k] = t2;
                B[j + 2][k] = t3; B[j + 3][k] = t4;

                B[j][k + 4] = t5; B[j + 1][k + 4] = t6;
                B[j + 2][k + 4] = t7; B[j + 3][k + 4] = t8;
            }
            for (p = j; p < j + 4; p++)
            {
                t1 = A[i + 4][p]; t2 = A[i + 5][p];
                t3 = A[i + 6][p]; t4 = A[i + 7][p];
                t5 = B[p][i + 4]; t6 = B[p][i + 5];
                t7 = B[p][i + 6]; t8 = B[p][i + 7];
                B[p][i + 4] = t1; B[p][i + 5] = t2;
                B[p][i + 6] = t3; B[p][i + 7] = t4;
                B[p + 4][i] = t5; B[p + 4][i + 1] = t6;
                B[p + 4][i + 2] = t7; B[p + 4][i + 3] = t8;
            }
            for (k = i + 4; k < i + 8; k++)
            {
                t1 = A[k][j + 4]; t2 = A[k][j + 5];
                t3 = A[k][j + 6]; t4 = A[k][j + 7];
                B[j + 4][k] = t1; B[j + 5][k] = t2;
                B[j + 6][k] = t3; B[j + 7][k] = t4;
            }
        }
    }
```

这段代码达到了要求。

对于 61×67 的情况，其行数和列数都不是 8 的倍数，因此只能随机分块。

我尝试了按 10*10 分块、11*11 分块、12*12 分块，一直到 20*20 分块，将每种分块方式注册一个函数后测试，发现有多种分块方式满足要求。测试结果如下图。

```
Function 4 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 4 (trans3): hits:6122, misses:2057, evictions:2025

Function 5 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 5 (trans4): hits:6131, misses:2048, evictions:2016

Function 6 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 6 (trans5): hits:6183, misses:1996, evictions:1964

Function 7 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 7 (trans6): hits:6158, misses:2021, evictions:1989

Function 8 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 8 (trans7): hits:6187, misses:1992, evictions:1960

Function 9 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 9 (trans8): hits:6229, misses:1950, evictions:1918

Function 10 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 10 (trans9): hits:6218, misses:1961, evictions:1929

Function 11 (12 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 11 (trans10): hits:6200, misses:1979, evictions:1947
```

最后我选择了我编写的 trans10 函数作为 submission，其分块方式是 19*19 分块，代码如下图：

```
char trans10_desc[] = "trans10";
void trans10(int M, int N, int A[N][M], int B[M][N])
{
    int STRIDE = 19;
    if (M == 61 && N == 67)
    {
        for (int i = 0; i < N; i += STRIDE)
        {
            for (int j = 0; j < M; j += STRIDE)
            {
                for (int k = i; k < i + STRIDE && k < N; k++)
                {
                    for (int p = j; p < j + STRIDE && p < M; p++)
                    {
                        int t = A[k][p];
                        B[p][k] = t;
                    }
                }
            }
        }
    }
}
```

32×32（10 分）：运行结果截图

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./test-trans -M 32 -N 32

Function 0 (1 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

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

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./test-trans -M 64 -N 64

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9066, misses:1179, evictions:1147

Summary for official submission (func 0): correctness=1 misses=1179

TEST_TRANS_RESULTS=1:1179
```

61×67（20 分）：运行结果截图

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab ./test-trans -M 61 -N 67

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6200, misses:1979, evictions:1947

Summary for official submission (func 0): correctness=1 misses=1979
```

两个实验任务总体评分(driver.py)：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab python 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        1179
Trans perf 61x67          10.0        10        1979
        Total points      53.0        53
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab6/cachelab
```

第 4 章 总结

4.1 请总结本次实验的收获

- 深入理解了 cache 的机理
- 手工模拟了 cache，学会了 cache 底层逻辑的设计，尤其是 LRU 算法
- 通过矩阵转置的例子，学会了编写面向内存友好的程序代码

4.2 请给出对本次实验内容的建议

- 在矩阵转置的实验中，部分得分点卡的比较严，例如如果不强制使用全局变量，使得编译器使用寄存器传递变量的话，就过不了关，希望老师能在这一点上把实验要求放宽松一点点。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.

[7] https://www.cnblogs.com/andashu/p/6378000.html

[8] https://blog.csdn.net/weixin_43821874/article/details/86481338