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

学　　号 1170300825

班　　级 1703008

学　　生 李大鑫

指 导 教 师 郑贵滨

实 验 地 点

实 验 日 期

计算机科学与技术学院

目 录

第 1 章 实验基本信息 ..... - 3 -

1.1 实验目的 ..... - 3 -
1.2 实验环境与工具 ..... - 3 -
1.2.1 硬件环境 ..... - 3 -
1.2.2 软件环境 ..... - 3 -
1.2.3 开发工具 ..... - 3 -
1.3 实验预习 ..... - 3 -

第 2 章 实验预习 ..... - 5 -

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） ..... - 5 -
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） ..... - 5 -
2.3 写出各类 CACHE 的读策略与写策略（5 分） ..... - 6 -
2.4 写出用 GPROF 进行性能分析的方法（5 分） ..... - 7 -
2.5 写出用 VALGRIND 进行性能分析的方法（（5 分） ..... - 8 -

第 3 章 CACHE 模拟与测试 ..... - 10 -

3.1 CACHE 模拟器设计 ..... - 10 -
3.2 矩阵转置设计 ..... - 13 -

第 4 章 总结 ..... - 19 -

4.1 请总结本次实验的收获 ..... - 19 -
4.2 请给出对本次实验内容的建议 ..... - 20 -

参考文献 ..... - 21 -

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；GDB/OBJDUMP；DDD/EDB 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

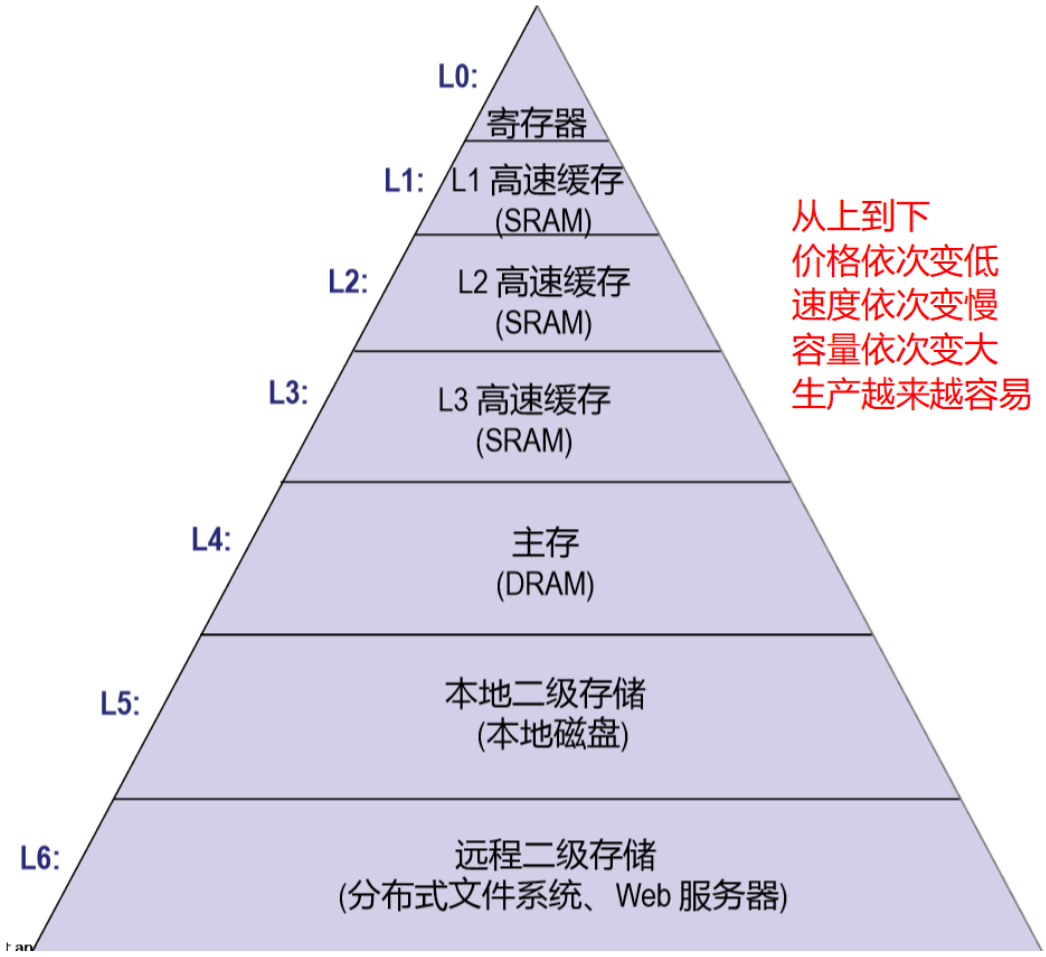
■ 用 **CPUZ** 等查看你的计算机 **Cache** 各参数，写出 **Cache** 的基本结构与参数：**C S E B s e b**

■ 写出各类 **Cache** 的读策略与写策略

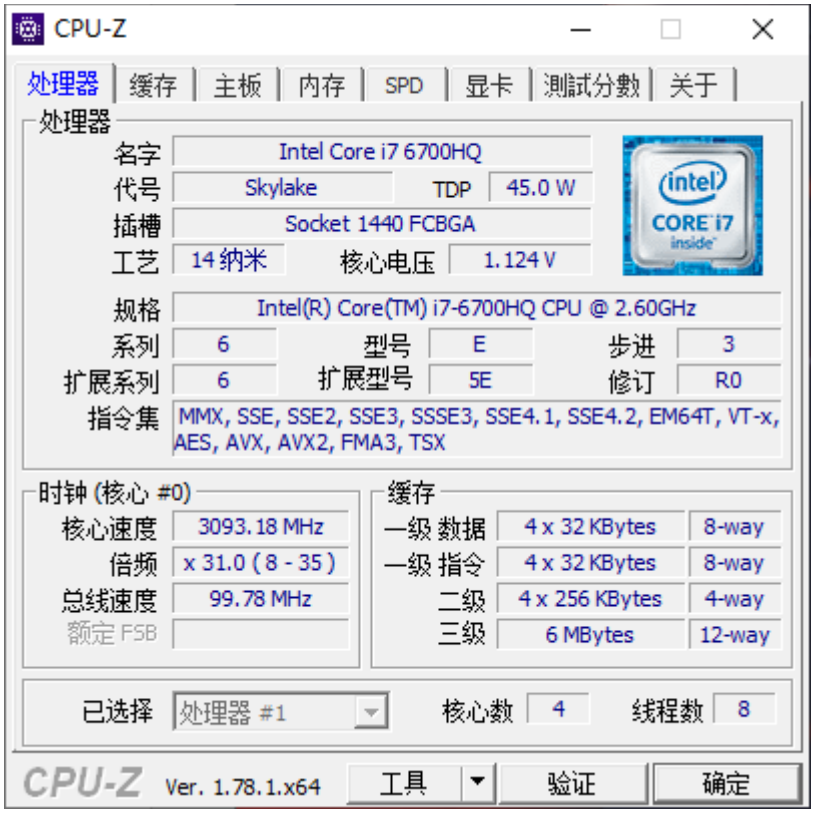
■ 掌握 **Valgrind** 与 **Gprof** 的使用方法

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）



	C	S	E	B	s	e	b
L1	4x32 KBytes	64	8	64	6	3	6
L2	4x256Kbytes	2^18	4	64	18	2	6
L3	6Mbytes	2^13	12	64	13	$\log_2 12$	6

2.3 写出各类 Cache 的读策略与写策略（5 分）

读策略：读数据需要传递一个 addr，每个存储器有 m 位，所以 addr 的地址位数一共有 M=2^m 位，addr 分为三部分：t 位标记 tag、s 位组索引、b 位块偏移。在缓存中进行寻址的时候，首先通过 s 位组索引定位映射到的组号，然后在组内通过查找 tag 位匹配且同时 valid 为 1 的缓存行，最后通过快偏移定位需要的数据在行内的位置。如果命中返回，如果没有发生命中，则采用 LRU 策略进行驱逐和替换。

写策略：采用直写和写回两种方式。直写就是立即将 w 的高速缓存块写回到

紧接着的低一层中；写回就是尽可能地推迟更新，只有当替换算法要驱逐这个更新过的块时，才把他写到紧接着的低一层中。发生不命中时，采用两种方案：一种是写分配，而另一种是非写分配。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；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只能profile用户态的函数，对应系统调用的函数，gprof不能profile。

使用gprof 只需在编译的时候 加上-pg参数就行了。下面为main.c文件的内容。编译 gcc -pg main.c -o main 生成main执行文件

```
#include <stdlib.h>

static unsigned long sum(int num)
{
    int ret = 0;
    for(int i = 0; i < num; i++)
           ret += i;
    return ret;
}

static unsigned long fb(int num)
{
    if(num < 2)
       return 1;
    else
       return fb(num -1) + fb(num-2);
}

int main(int argc, char** argv)
{
    if(argc < 2)
    {
       printf("usage num\n");
       return 1;
    }
    int num = atoi(argv[1]);
    unsigned long r1 = sum(num);
    unsigned long r2 = fb(num);
    printf("r1=%ld, r2=%ld\n", r1, r2);
    return 0;
}
```

运行./main结束后 会在当前目录生成gmon.out的文件。

我们执行gprof ./main 就会输出main的profile，不过这样并不太直观。我们现在可以用工具把profile数据图形化出来。

gprof ./main > profile.txt 把数据输出到profile.txt文件中
2)gprof2dot.py profile.txt > profile.dot 生成dot文件
3)dot -Tsvg -o gprof.svg 生成svg文件 我们就直接用浏览器就可以打开svg看那个函数是热点了。

gprof2dot.py脚本可以用githun上fork下来，dot工具，linux可以直接安装。centos命令 yum install graphviz。其他发行版本的，把安装命令换一下就行了。

当然也可以直接一步 gprof ./main| gprof2dot.py -n0 -e0 | dot -Tpng -o out.png生成png文件

更详细的profile图
gprof -p -q ./main| gprof2dot.py -n0 -e0 | dot -Tpng -o out.png

gprof的一下参数

-m num或--min-count=num：不显示被调用次数小于num的函数；

-p 只输出函数的调用图

-q 只输出函数的时间消耗列表

2.5 写出用 Valgrind 进行性能分析的方法（（5 分）

Valgrind的主要作者Julian Seward获得了2006年的Google-O'Reilly开源大奖之一——Best Tool Maker
valgrind的主要功能
Valgrind工具包包含多个工具：
Memcheck：
检查程序中的内存问题，包括使用未初始化的内存；使用已经释放的内存；使用内存越界；对堆栈的非法访问；内存泄露；malloc/free/new/delete申请和释放内存的匹配等内存问题
callgrind：
Callgrind收集程序运行时的一些数据，函数调用关系等信息，还可以有选择地进行cache 模拟。在运行结束时，它会把分析数据写入一个文件。callgrind_annotate可以把这个文件的内容转化成可读的形式。
cachegrind：
它模拟CPU中的一级缓存I1,D1和L2二级缓存，能够精确地指出程序中 cache的丢失和命中。如果需要，它还能够为我们提供cache丢失次数，内存引用次数，以及每行代码，每个函数，每个模块，整个程序产生的指令数。这对优化程序有很大的帮助。
helgrind：

它主要用来检查多线程程序中出现的竞争问题。Helgrind 寻找内存中被多个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步的地方，而且会导致难以发掘的错误。

massif：

堆栈分析器，它能测量程序在堆栈中使用了多少内存，告诉我们堆块，堆管理块和栈的大小。Massif能帮助我们减少内存的使用，在带有虚拟内存的现代系统中，它还能够加速我们程序的运行，减少程序停留在交换区中的几率。

valgrind的安装

1、 到www.valgrind.org下载最新版valgrind-3.2.3.tar.bz2
2、 解压安装包：tar –jxvf valgrind-3.2.3.tar.bz2
3、 解压后生成目录valgrind-3.2.3
4、 cd valgrind-3.2.3
5、 ./configure
6、 make;make install

valgrind的使用

用法: valgrind --tool=tool_name [options] program_name:

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

常用选项，适用于所有Valgrind工具

-tool= 最常用的选项。运行 valgrind中名为toolname的工具。默认memcheck。
--h 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。
--–quiet 安静地运行，只打印错误信息。
--–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= 输出LOG到描述符文件 [2=stderr]
--log-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=输出LOG信息到 file
--log-file-qualifier= 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 显示多少个函数栈 [12]
--error-limit=no|yes 如果太多错误，则停止显示新错误? [yes]
--error-exitcode= 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= 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|yes| 要求对leak给出详细信息? [yes]

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：

因为程序已经给出了用户交互相关的代码，所以我们的主要工作就是补全 initCache，freeChache，accessData。

initData：Init 中负责初始化 cache 数组，给 cache 动态开辟空间，并将 valid，tag，lru 赋值为 0。需要注意的是开辟空间的大小。如下：

```
void initCache() {
    int i, j;
    cache = (cache_t)malloc(sizeof(cache_set_t) * S);
    for (i = 0; i < S; i++) {
        cache[i] = (cache_set_t)malloc(sizeof(cache_line_t) * E);
        for (j = 0; j < E; j++) {
            cache[i][j].valid = 0;
            cache[i][j].tag = 0;
            cache[i][j].lru = 0;
        }
    }
}
```

freeCache：只需要将 cache 数组 free 即可。

accessData：首先通过位运算取出 addr 的 s 位组索引和 tag。将 lru_count 定义为一个时间戳，每次在缓存中寻找都会使 lru_count++，这样将每一个在缓存中新建的 cache_line 的 lru 都设置为 lru_count 并且每次访问缓存如果命中也更新 lru 为 lru_count 的话就可以使每一个位于缓存的 cache_line 都记录着自己的访问时间 lru。

访问情况分为三种：1）命中：更新时间戳 2）缓存不命中且冲突：选择 lru 最小的即据现在访问时间最远的一个 cache_line 进行覆盖 3）缓存不命中且不冲突：此时在第一个 valid=0 即为空的 cache_line 中存放“在下级缓存读取到的”新 cache_line。

如下：

```
void accessData(mem_addr_t addr)
{
  unsigned long long int mask;
  int i, j;
  int flag = 0, flag2 = -1;
  // 构造mask 取得s位组索引
  // 分情况
  mask = 1;
  mask = (mask<<s)-1;
  unsigned long long int mys = (addr >> b) & mask;

  mask = 1;
  mask =(mask<<(b+s))-1;
  unsigned long long int mytag = (addr >> (b + s)) & mask;

  unsigned long long int treasure = cache[mys][0].lru, treasureblock =
0;
  lru_counter++;
  for (i = 0; i < E; i++) {
      if (cache[mys][i].valid == 1 && cache[mys][i].tag == mytag) {
          hit_count = hit_count + 1;
          flag = 1;
          cache[mys][i].lru = lru_counter;
          break;
      }
      else if (cache[mys][i].valid == 0 && flag2 == -1)
          //一个valid是0的索引
          flag2 = i;
  }
  if (flag == 0) {                      //没有命中
      miss_count = miss_count + 1;
      if (flag2 == -1) {                 //都是满的  此时是冲突的状态
          for (j = 0; j < E; j++) {
              if (treasure > cache[mys][j].lru) {
                  treasure = cache[mys][j].lru;
                  treasureblock = (unsigned long long)j;
              }
          }
          //trash存储着lru count最小的cache_line
          //treasure存储着lru_count最大的cache_line
          //冲突 覆盖
          //LRU 策略：覆盖距现在访问时间最远的cache_line
          cache[mys][treasureblock].tag = mytag;
          cache[mys][treasureblock].valid = 1;
          cache[mys][treasureblock].lru = lru_counter;
          eviction_count = eviction_count + 1;
      }
      else {                                                  //此时有空的 也就不
会产生冲突 放置策略：放在第一个valid位是 0 的cache_line
          cache[mys][flag2].valid = 1;
          cache[mys][flag2].tag = mytag;
          cache[mys][flag2].lru = lru_counter;  //lru=1
      }
```

```
}
  }
```

测试用例 1 的输出截图（5 分）：

```
linda: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）：

```
linda:cachelab-handout>./csim -s 4 -E 2 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:2
```

测试用例 3 的输出截图（5 分）：

```
linda:cachelab-handout>./csim -s 2 -E 1 -b 4 -t traces/dave.trace
hits:2 misses:3 evictions:1
```

测试用例 4 的输出截图（5 分）：

```
linda:cachelab-handout>./csim -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）：

```
linda:cachelab-handout>./csim -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）：

```
linda:cachelab-handout>./csim -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）：

```
linda:cachelab-handout>./csim -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）：

```
linda:cachelab-handout>./csim -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
```

test-csim 运行结果：

```
linda:cachelab-handout>gcc -o csim csim.c cachelab.c  cachelab.h
linda: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
```

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

3.2 矩阵转置设计

提交 trans.c

程序设计思想：

据重要信息，（s,E,b）=(5,1,5)可以推断出 cache 是 32x32 字节的，其中一组一行，一共 8B 可以存放 8 个 int，所以 cache 中最多可以存放 8 行的 32x32 的 int 矩阵。

在 B[i][j]=A[i][j]的过程中，A 矩阵经历一个读操作，B 矩阵经历一个读操作一个写操作。所以这条语句会产生两次缓存查找。

1） 首先分析 32x32 的情况：我们把一个 8 个 int 当做一个 block，把 8x8 的 int 矩阵当做一个 blocking，进行矩阵转置，一个 block 代表一行 cache 可以存储的大小，一个 blocking 代表每次循环进行转置操作的单位大小。

首先看一下组号的映射情况，因为 cache 最多只能够存放 8 行的 int 矩阵,所以映射后的组号是 block 的序号取模 32。当两个 block 映射之后的组号相同的时候就会产生冲突，这时就应该产生 cache 中的数据替换。因为 AB 在内存中都是连续放的，而且其大小都是 cache 的整数倍，所以两个矩阵的组号映射情况应该是相同的。也就是 A[i][j]和 B[i][j]射到相同的组号。

以 blocking 为单位进行转置，对于不在对角线上的 int，进行转置的

时候，A 读数据的时候每隔 8 个产生一次 miss，B 读数据的时候前 8 次每次都 miss，后面的 24 个 int 都会 hit，所以两者的命中率都是 1/8（这个画图可以看出来）。

对于 i=j，int 数据位于矩阵对角线的情况。A[i][j]与 B[i][j]映射之后映射到相同的组，所以会产生冲突需要对 cache 进行替换。替换之后由于还要对 A 进行读操作，所以还是需要在 cache 中替换回 A 数组中的 block，这就产生了额外开销。

如何消除这种替换呢？遇到是矩阵对角的情况我们可以先选择保存 A 中的值和行号到中间变量，等当前 block 的循环结束之后我们再将值赋给 B。

代码如下：

```
if (N == 32)
    {
        Bsize = 8;
        for(colBlock = 0; colBlock < N; colBlock += 8)
        {
            for(rowBlock = 0; rowBlock < N; rowBlock += 8)
                {
                    for(r = rowBlock; r < rowBlock + 8; r++)
                    {
                        for(c = colBlock; c < colBlock + 8; c++)
                        {
                            if(r != c)
                            {
                                B[c][r] = A[r][c];
                            } else
                            {
                                temp = A[r][c];
                                d = r;
                            }
                        }
                        if (rowBlock == colBlock)
                        {
                            B[d][d] = temp;
                        }
                    }
            }
        }
    }
```

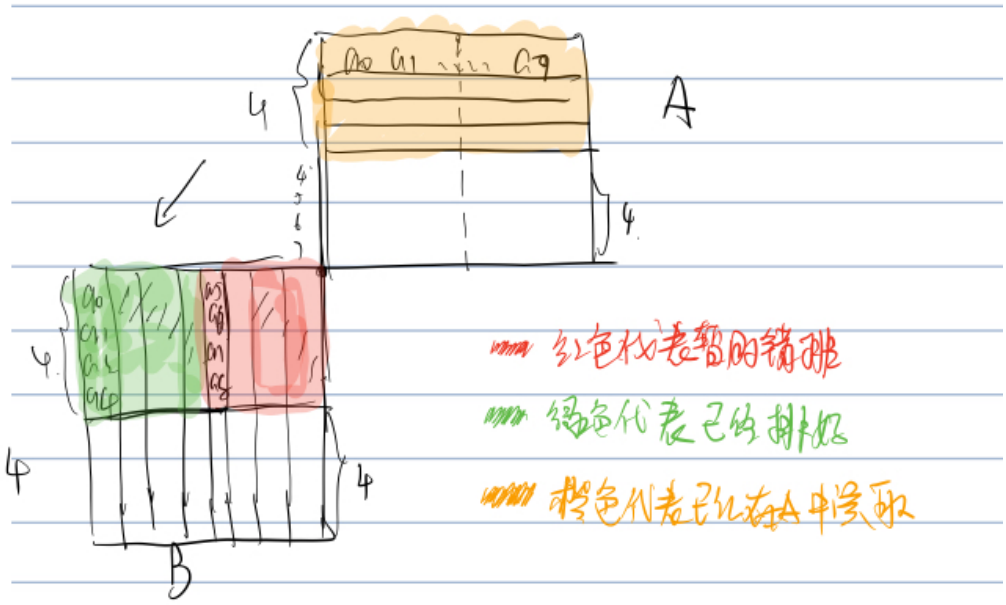
2） 64x64

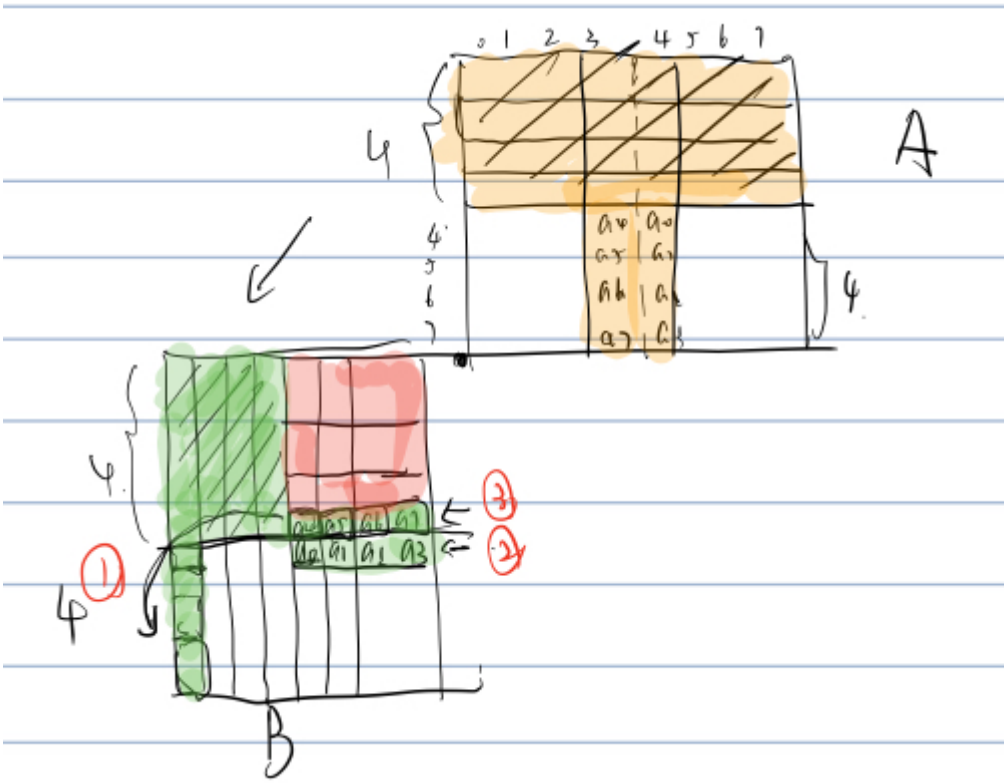
首先考虑，**cache 中只能放下 4 行 64x64int 数组数据**，如果依然使用

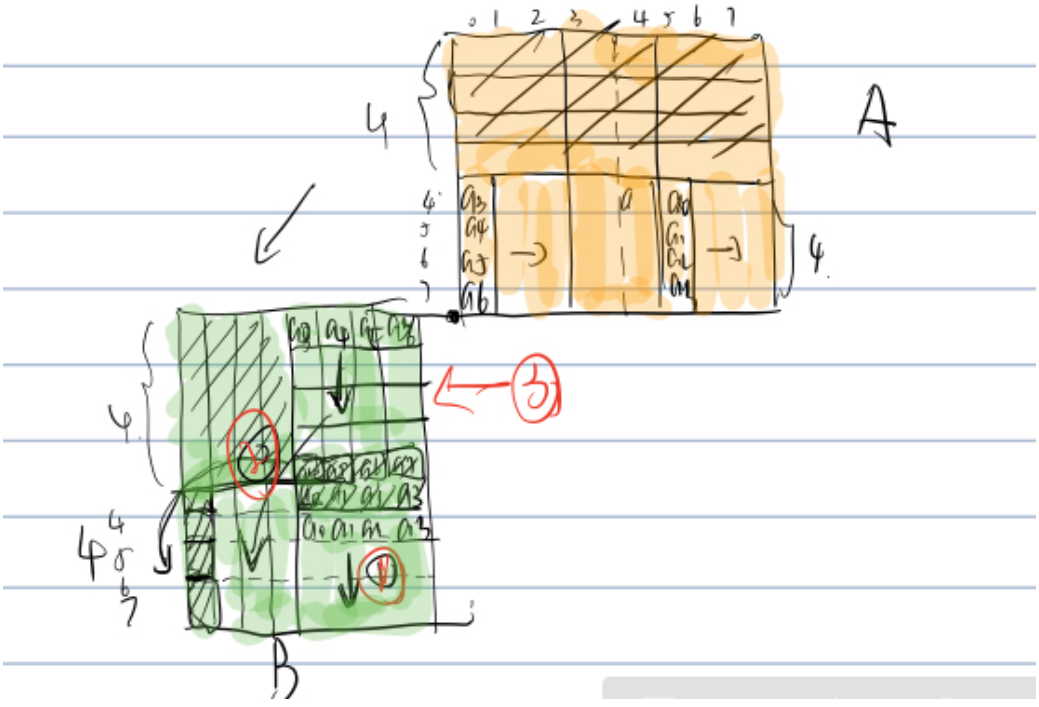
8x8 的 blocking，在一次 blocking 中就会导致 B 中后四行 block 与 A 中前四行 block 发生 cache 替换冲突。4x4 的 blocking 优化之后依然不能够满足题目要求。我们考虑依然使用 8x8 的 blocking，但是需要解决替换冲突的问题。

我们可以使用B数组空闲位置作为缓存同时将B数组的读写操作和A数组的读操作进行分离解决这种冲突。

操作分离可以使用中间变量完成。对于一个 8x8blocking 而言，一次循环我们先考虑利用 8 个中间变量进行来进行 4x8 的转置，但是对于一些 int 我们不能直接将他们放在正确的位置上，我们首先需要利用前四行后四列作为缓存来存放本应该放在后四行的数据，然后恢复正确位置，进行后四行的操作，具体操作如下图：







代码如下：

```
void transpose_64_64(int M, int N, int A[N][M], int B[M][N]) {
    int colRun, rowRun, k, a0, a1, a2, a3, a4, a5, a6, a7;

    for(colRun=0; colRun<64; colRun+=8 ){
        for(rowRun=0; rowRun<64; rowRun+=8 ){
        for(k=0; k<4; k++){
            a0 = A[colRun+k][rowRun+0];
            a1 = A[colRun+k][rowRun+1];
            a2 = A[colRun+k][rowRun+2];
            a3 = A[colRun+k][rowRun+3];
            a4 = A[colRun+k][rowRun+4];
            a5 = A[colRun+k][rowRun+5];
            a6 = A[colRun+k][rowRun+6];
            a7 = A[colRun+k][rowRun+7];

            B[rowRun+0][colRun+k+0] = a0;
            B[rowRun+0][colRun+k+4] = a5;
            B[rowRun+1][colRun+k+0] = a1;
            B[rowRun+1][colRun+k+4] = a6;
            B[rowRun+2][colRun+k+0] = a2;
            B[rowRun+2][colRun+k+4] = a7;
            B[rowRun+3][colRun+k+0] = a3;
            B[rowRun+3][colRun+k+4] = a4;
        }

        a0 = A[colRun+4][rowRun+4];
        a1 = A[colRun+5][rowRun+4];
        a2 = A[colRun+6][rowRun+4];
        a3 = A[colRun+7][rowRun+4];
        a4 = A[colRun+4][rowRun+3];
        a5 = A[colRun+5][rowRun+3];
        a6 = A[colRun+6][rowRun+3];
        a7 = A[colRun+7][rowRun+3];


        B[rowRun+4][colRun+0] = B[rowRun+3][colRun+4];
        B[rowRun+4][colRun+4] = a0;
        B[rowRun+3][colRun+4] = a4;
        B[rowRun+4][colRun+1] = B[rowRun+3][colRun+5];
        B[rowRun+4][colRun+5] = a1;
        B[rowRun+3][colRun+5] = a5;
        B[rowRun+4][colRun+2] = B[rowRun+3][colRun+6];
        B[rowRun+4][colRun+6] = a2;
        B[rowRun+3][colRun+6] = a6;
        B[rowRun+4][colRun+3] = B[rowRun+3][colRun+7];
        B[rowRun+4][colRun+7] = a3;
        B[rowRun+3][colRun+7] = a7;

        for(k=0;k<3;k++){
            a0 = A[colRun+4][rowRun+5+k];
            a1 = A[colRun+5][rowRun+5+k];
            a2 = A[colRun+6][rowRun+5+k];
            a3 = A[colRun+7][rowRun+5+k];
            a4 = A[colRun+4][rowRun+k];
            a5 = A[colRun+5][rowRun+k];
            a6 = A[colRun+6][rowRun+k];
            a7 = A[colRun+7][rowRun+k];
```

```
            B[rowRun+5+k][colRun+0] = B[rowRun+k][colRun+4];
            B[rowRun+5+k][colRun+4] = a0;
            B[rowRun+k][colRun+4] = a4;
            B[rowRun+5+k][colRun+1] = B[rowRun+k][colRun+5];
            B[rowRun+5+k][colRun+5] = a1;
            B[rowRun+k][colRun+5] = a5;
            B[rowRun+5+k][colRun+2] = B[rowRun+k][colRun+6];
            B[rowRun+5+k][colRun+6] = a2;
            B[rowRun+k][colRun+6] = a6;
            B[rowRun+5+k][colRun+3] = B[rowRun+k][colRun+7];
            B[rowRun+5+k][colRun+7] = a3;
            B[rowRun+k][colRun+7] = a7;
        }
        }
    }
}
```

3） 61x67

思想和 1）相近，使用 16x16 的 blocking，同时也特殊处理对角线。

代码如下：

```
Bsize = 16;
        for (colBlock = 0; colBlock < M; colBlock += Bsize)
        {
            for (rowBlock = 0; rowBlock < N; rowBlock += Bsize)
            {
                for(r = rowBlock; (r < N) && (r < rowBlock + Bsize); r++)
                {
                    for(c = colBlock; (c < M) && (c < colBlock + Bsize); c++)
                    {
                        if (r != c)
                        {
                            B[c][r] = A[r][c];
                        } else
                        {
                            temp = A[r][c];
                            d = r;
                        }
                    }
                    if(rowBlock == colBlock)
                    {
                        B[d][d] = temp;
                    }
                }
            }
        }
```

32×32（10 分）：运行结果截图

```
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

```
linda: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:8946, misses:1299, evictions:1267

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=1299

TEST_TRANS_RESULTS=1:1299
```

61×67（20 分）：运行结果截图

```
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:6373, misses:1809, evictions:1777

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=1809

TEST_TRANS_RESULTS=1:1809
```

第 4 章 总结

4.1 请总结本次实验的收获

* 简单缓存区的实现原理
* LRU 原则
* 对二位矩阵操作利用缓存的性质进行代码优化

4.2 请给出对本次实验内容的建议

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.