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 1170500820

班　　级 1703008

学　　生 周述哲

指 导 教 师 郑贵滨

实 验 地 点 G712

实 验 日 期 2018/11/29

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

微星 GE62MVR 笔记本电脑

1.2.2 软件环境

VMware，valgrind，vim

1.2.3 开发工具

VMware

1.3 实验预习

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）

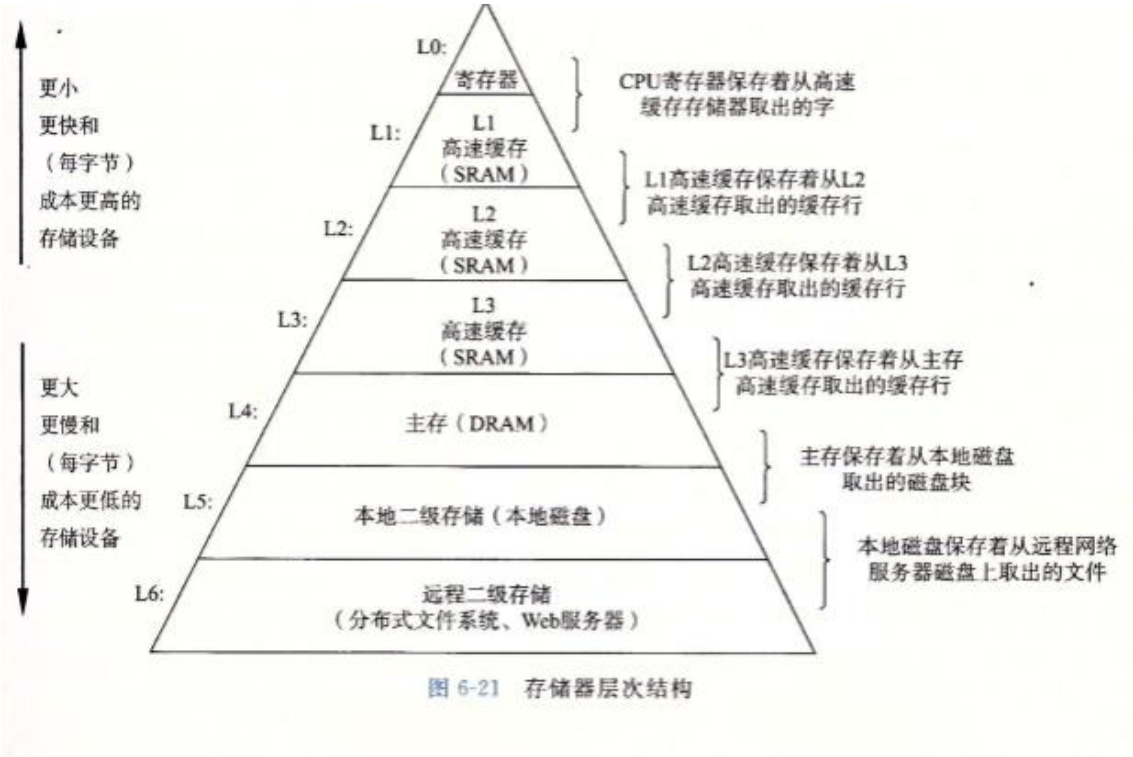


图 6-21 存储器层次结构

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）

L1 数据 cache :32Kbytes ×4 8-way set associative, 64-byte line size

C=32768=32KBytes S=4 E=8 B=1024 s=2 e=3 b=10

L1 指令 cache :32Kbytes ×4 8-way set associative, 64-byte line size

C=32768=32KBytes S=4 E=8 B=1024 s=2 e=3 b=10

L2 cache :256Kbytes ×4 4-way set associative, 64-byte line size

C=32768=256KBytes S=64 E=4 B=1024 s=6 e=2 b=10

L3 cache :6Mbytes ×4 12-way set associative, 64-bbyte line size

C=32768=32KBytes S=2730 E=12 B=1024 s=log2(2730) e=log2(12) b=10

2.3 写出各类 Cache 的读策略与写策略（5 分）

读策略：

当指令要求从内存中读取一个值的时候，cpu 会先到最高级 cache 中去寻找是否有包含该内存字块的行，如果有则命中，将内容返回给 cpu。

若没有，cpu 就会到下一个层级的 cache 中去寻值，依次类推，直到找到为止。找到后会将该内容所在内存块存储到最高速缓存行中。

写策略：

写策略有两种，

一种叫做直写，假设要写一个已经缓存了的内容，在高速缓存更新了其副本之后，立即将该内容的高速缓存块直接写回到紧接着的低一层中去。这个方法简单易实现，但是每次写都会引起总线流量。

另一种方法叫做写回：尽可能的推迟更新，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，

而在处理写不命中时，又有两种方法

一种称为写分配，即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

另一种方法称为非写分配，避开高速缓存，直接将这个内容写到低一层中。

2.4 写出用 gprof 进行性能分析的方法（5 分）

gprof 是 GNU profiler 工具。可以显示程序运行的“flat profile”，包括每个函数的调用次数，每个函数消耗的处理器时间。也可以显示“调用图”，包括函数的调用关系，每个函数调用花费了多少时间。还可以显示“注释的源代码”，是程序源代码的一个复本，标记有程序中每行代码的执行次数。

其基本使用方法如下：

1 使用-pg 选项编译和链接程序。

2. 运行程序，使之运行完成后生成供 gprof 分析的数据文件（默认是 gmon.out）。

3 使用 gprof 程序分析程序生成的数据。

4. 可用 python 生成 png 图来直观看。

2.5 写出用 Valgrind 进行性能分析的方法（（5 分）

valgrind 的几个基本操作如下：

1，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。这也是本文将重点介绍的部分。

2，Callgrind。它主要用来检查程序中函数调用过程中出现的问题。

3，Cachegrind。它主要用来检查程序中缓存使用出现的问题。

4，Helgrind。它主要用来检查多线程程序中出现的竞争问题。

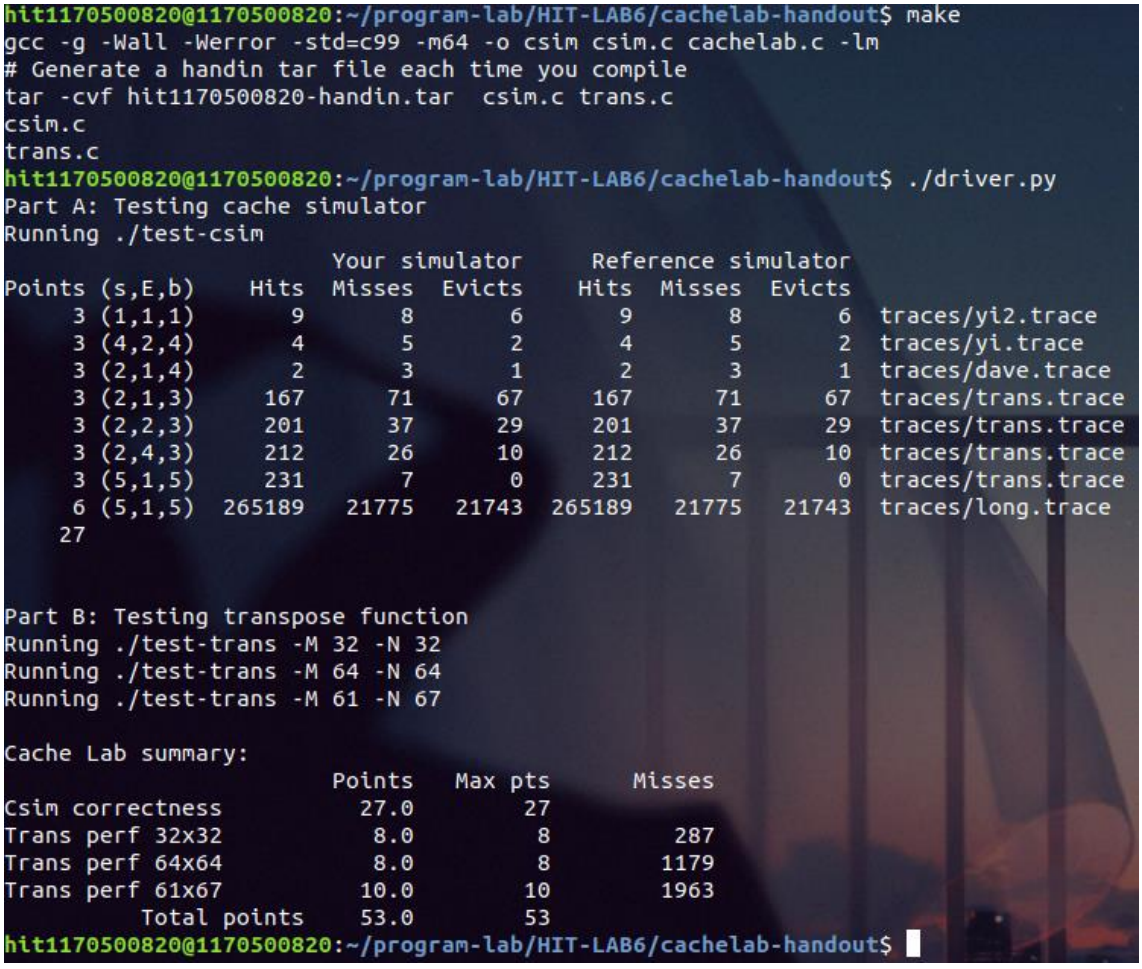
5，Massif。它主要用来检查程序中堆栈使用中出现的问题。

6，Extension。可以利用 core 提供的功能，自己编写特定的内存调试工具

第 3 章 Cache 模拟与测试

3.1 Cache 模拟器设计

提交 csim.c



csim.c 源程序随文档一起上交

程序设计思想：

本程序的设计目的是要模拟计算机的对内存访问的访存处理机制。

cache 的基本工作原理就是在程序对内存进行访问时，不仅访问程序指定的内存，同时还将对应内存的一片区域的数据都抓取到 cache 中，依赖于程序执行

的时间/空间局部性的原理，下一次程序访问内存时，要访问的数据已经放在 cache 里的概率较大，以这种机制来提高程序的运行速度。

cache 的基本单元是行，每一行中含有一次抓取的数据块，对应的地址标记位以及判定是否有效的有效位。多行组成一组，一级 cache 由多个组组成。

通过结构体数组来模拟 cache 的存储结构：

```
typedef struct cache_line {
    char valid;
    mem_addr_t tag;
    unsigned long long int lru;//lru：将最近最少使用的代码数据置换出去
} cache_line_t;//定义了一个高速缓冲器的内容
```

在这个定义中，valid 是有效位，tag 则是判定是否含有程序所要访问的数据的地址标记位。而 lru 则是实现”当需要载入新的内存数据时，优先将最近最少使用的数据驱逐”的机制的一个判定数据，关于这个机制，后面再详细阐述。

首先观察主函数，推断各个函数功能

```
/*
 * main - Main routine
 */
int main(int argc, char* argv[])
{
    char c;

    while( (c=getopt(argc,argv,"s:E:b:t:vh")) != -1){ ... }

    /* Make sure that all required command line args were specified */
    if (s == 0 || E == 0 || b == 0 || trace_file == NULL){ ... }

    /* Compute S, E and B from command line args */
    //sEb已知
    S = (int)pow(2, (double)s);
    B = (int)pow(2, (double)b);


    /* Initialize cache */
    initCache();

#ifdef DEBUG_ON 非活动预处理器块
#endif

    replayTrace(trace_file);

    /* Free allocated memory */
    freeCache();

    /* Output the hit and miss statistics for the autograder */
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}
```

main 函数中先从参数输入行读取用来规划 cache 大小的 s,E,b 数据以及使用的访存轨迹文件，其后对指令的解析以及对文件的读写以及由老师给出的框架函数完成了。

我们所需要补充的函数/功能主要有以下几个部分：

1， `initCache();`函数，用来根据 seb 数据对 cache 进行初始化。

2， `freeCache();`函数，用于释放在 initCache 函数中为 cache 分配的内存

3， `void accessData(mem_addr_t addr)`，该函数用于根据输入的访存轨迹，在 cache 中进行读写，以模拟真实情况下 cache 的工作过程

4， 将输入的 s 和 b 转化为 S 和 B

首先进行 s，b 的转化，根据 cache 结构，只需要简单的计算 2 次幂就能得出结果：

```
S = (int)pow(2, (double)s);
B = (int)pow(2, (double)b);
```

initcache 函数的实现：

cache 由 S 组，每组 E 行组成，所以需要定义一个以 cache 行为基本元素的二维数组。只需要简单的用两个 for 循环进行初始化即可，其中注意要将 cache 结构体中的每个元素都初始化为 0.

freeCache 函数的实现：

由于 free 是对地址进行 free，所以依旧是用 for 循环来遍历 cache 的每个 malloc 过的地址即可，这个函数只需要按照 initCache 函数来对应的写就能够简单实现。

```
void freeCache()
{
//todo...
    int i, j;
    for (i = 0; i < S; i++)
        free(cache[i]);
    free(cache);

}
```

accessData 函数的实现：

accessData 比较复杂，首先按照地址的组索引定位到 cache 的组，然后在对应的组中进行搜索，总共需要处理三种情况：

1， 如果需要访问的内存地址在 cache 中有，记一次 hit。

2， 如果需要访问的内存地址在 cache 中不存在，但同时 cache 中有些行的有效位为 0，也就是还有空行，那么就直接将数据载入到空行即可，此时记一次 miss。

3， 如果如果需要访问的内存地址在 cache 中不存在，而 cache 又不存在无效位，也就是所有的行都装载了数据，那么就需要有选择的将其中某一行的数据进行驱逐，然后载入新的数据，记一次 miss 和一次 eviction。

在上述的第三点中，驱逐数据的选择机制就是 lru 机制，也就是如果一个内存地址在最近才被访问过，那么更有可能继续被访问，则保留。如果一个内存地址很久一起访问过，那么程序倾向于不会再访问它，这样的数据就会被驱逐。而这个机制就是通过 cache 结构体中的 lru 元素来实现。

为了定量的比较这种性质，需要再写一个新的函数：

```
void release_lru()
{
    int i, j;
    for (i = 0; i < S; i++)
        for (j = 0; j < E; j++)
            if (cache[i][j].tag)
                cache[i][j].lru++;
}
```

每次内存访问时，都将 cache 结构体内的元素累加，代表访问时间的流逝。

而每一个被访问数据中的 lru 位都进行清零，以表示该数据刚刚被访问过或被更新过。

这样，lru 越大，代表该数据越久没被访问；lru 越小，代表该数据被访问的时间越近。

接下来开始实现 accessData 函数的主体功能。

首先将组索引和行标记值计算出来：

```
group_index = (addr&(set_index_mask << b)) >> b;
input_tag = addr >> (b + s);
```

组索引和行标记都是以位区分的，所以使用移位操作来将其计算出来。

接下来首先对 cache 是否命中进行判断：

```
for (j = 0; j < E;j++)
{
    if (!cache[group_index][j].valid)
        not_full = 1;
    if (cache[group_index][j].valid&&(cache[group_index][j].tag == input_tag))
    {
        hit_count++;
        cache[group_index][j].lru=0;
        hitted = 1;
        break;
    }
}
if (hitted&&verbosity)
    printf("hit ");
```

按照组索引定位到组后，对组中的每一行进行遍历搜索，当搜索到行标记匹配的行时，代表 cache 命中，则 hit++，然后将 lru 位置为 0 代表刚刚被访问过。

在该函数中设了两个标记，hitted 用来表达 cache 是否命中，not_full 用来判断 cache 是否行满。这两个标记做完下一步不同分支的判据。

如过没有命中，则需要进一步处理：

如果行未满则补行，已满则驱逐

补行不需要遵循什么规则，填充任意一个空行都是等价的。

而驱逐则需要对 lru 位逐个判断，选出其中 lru 值最大，也就是经过最久时间没有被访问的地址进行驱逐。

实现代码如下：

```
if (!hitted)
{
    if (verbosity)
    printf("miss ");
    miss_count++;
    if (not_full)
    {
        for (j = 0; j < E; j++)
        {
            if (!cache[group_index][j].valid )
            {
                cache[group_index][j].valid = 1;
                cache[group_index][j].tag = input_tag;
                cache[group_index][j].lru =0;
                break;
            }
        }
    }
    else
    {
        if (verbosity)
        printf("eviction ");
        eviction_count++;
        for (j = 0; j < E; j++)//找到最大lru
        {
            if (cache[group_index][j].lru > max_lru)
                max_lru = cache[group_index][j].lru;
        }
        for (j = 0; j < E; j++)
        {
            if (cache[group_index][j].valid&&(cache[group_index][j].lru == max_lru))
            {
                cache[group_index][j].lru = 0;
                cache[group_index][j].tag = input_tag;
                break;
            }
        }
    }
}
```

这就是 csim 程序的设计思想。

附：对 lru 的另一种更好的处理方法：

上述的对 lru 的使用方法是每一次访存时对所有 cache 块中的 lru 都进行累加，然后每次每块被访问到时则将其清零。

这种方法基于 lru 的思想就是随着时间推移，最近被访问过的程序下一个被访问到可能性要大于那些很久之前才被访问过的。

lru 的另一种实现方法就是定义一个全局的 lru_counter 当作计时器，每次读写操作时都对该计时器进行累加：

```
while( fgets(buf, 1000, trace_fp) != NULL)
    if(buf[1]=='S' || buf[1]=='L' || buf[1
        sscanf(buf+3, "%llx,%u", &addr, &l

        if(verbosity)
            printf("%c %llx,%u ", buf[1],

        accessData(addr);//读
        lru_counter++;

        /* If the instruction is R/W then
        if (buf[1] == 'M')
        {
            accessData(addr);//写(如果需要
            lru_counter++;
        }

        if (verbosity)
```

而每次访问某个数据时，只要使其 cache 块的 lru 值等于当前的 lru_counter 即可，这样同样能够实现对时间局部性的判断。

其实现代码如下:

```
void accessData(mem_addr_t addr)
{
    //todo...要计算掩码，还要累加计数变量
    //unsigned long long int max_lru = 0;
    unsigned long long int min_lru = 0xffffffffffffffff;
    int hitted = 0;
    int not_full = 0;
    mem_addr_t input_tag;
    mem_addr_t group_index;
    //group_index = ((addr << (ADDRESS_LENGTH - b - s)) >> (ADDRESS_LENGTH - s));
    group_index = (addr&(set_index_mask << b)) >> b;
    input_tag = addr >> (b + s);
    int j;
    for (j = 0; j < E; j++)
    {
        if (!cache[group_index][j].valid)
            not_full = 1;
        if (cache[group_index][j].valid && (cache[group_index][j].tag == input_tag))//如果位有效且标记位匹配
        {
            hit_count++;
            cache[group_index][j].lru = lru_counter;
            hitted = 1;
            break;
        }
    }
    if (hitted&&verbosity)
        printf("hit ");
    if (!hitted)
    {
        if (verbosity)
            printf("miss ");
        miss_count++;
        if (not_full)
        {
            for (j = 0; j < E; j++)
```

```
        {
            if (!cache[group_index][j].valid)
            {
                cache[group_index][j].valid = 1;
                cache[group_index][j].tag = input_tag;
                cache[group_index][j].lru = lru_counter;
                break;
            }
        }
    }
    else
    {
        if (verbosity)
            printf("eviction ");
        eviction_count++;
        for (j = 0; j < E; j++)//找到最小lru
        {
            if (cache[group_index][j].lru < min_lru)
                min_lru = cache[group_index][j].lru;
        }
        for (j = 0; j < E; j++)
        {
            if (cache[group_index][j].valid && (cache[group_index][j].lru == min_lru))//为最小lru对应的位置则驱逐原值
            {
                cache[group_index][j].lru = lru_counter;
                cache[group_index][j].tag = input_tag;
                break;
            }
        }
    }
}
```

测试用例 1 的输出截图（5 分）：

```
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
```

测试用例 2 的输出截图（5 分）：

```
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
```

测试用例 3 的输出截图（5 分）：

```
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
```

测试用例 4 的输出截图（5 分）：

```
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
```

测试用例 5 的输出截图（5 分）：

```
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
```

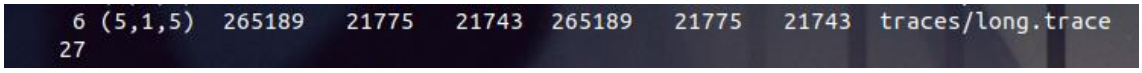
测试用例 6 的输出截图（5 分）：

```
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
```

测试用例 7 的输出截图（5 分）：

```
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
```

测试用例 8 的输出截图（10 分）：



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：

32×32 程序设计思想

首先根据实验指导 ppt 可知，矩阵转置实验的缓冲器的配置参数为：

s=5, E=1, b=5

故缓冲器的组数 S=2^5=32 每个高速缓存块的字节数 B=2^5=32 字节

每组只有一行。

每一个 int 类型的变量占用 4 字节，所以每个高速缓冲块一次能够存储连续的 8 个 int 类型的变量。考虑到可以通过充分利用缓冲块来对程序的缓冲性能进行优化，可以采取分块转置矩阵的方法。将一个 32×32 的矩阵分成 16 个 8×8 的矩阵来分别转置。尝试编写如下程序：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, tmp;
    int k, l;

    for (l = 0; l < 4; l++) {
        for (k = 0; k < 4; k++) {
            for (j = 8 * k; j < 8 * k + 8; j++) {
                for (i = 8 * l; i < 8 * l + 8; i++) {
                    tmp = A[i][j];
                    B[j][i] = tmp;
                }
            }
        }
    }
}
```

使用 test-trans 函数测试：

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

如果两个矩阵的缓冲块占用不会相互干扰的话，理论上来说，miss 的次数应当是：(8+8) ×16=256 次，因为每个 8×8 的分块矩阵在存取时都分别会 miss8 次，但是实际 miss 次数为 343 次与理论值不符，说明两个矩阵的缓冲块很可能有重合之处。对访存轨迹进行分析：

```
hit1170500820@1170500820:~/program-lab/HIT-LAB6/cachelab-handout$ ./csim-ref -v
 -s 5 -E 1 -b 5 -t trace.f0 >ref-515-trace-f0.txt
```

```
L 30a0fc,4 hit
S 34b000,4 miss eviction
L 30a17c,4 hit
S 34b004,4 hit
L 30a1fc,4 hit
S 34b008,4 hit
L 30a27c,4 hit
S 34b00c,4 hit
L 30a2fc,4 hit
S 34b010,4 hit
L 30a37c,4 hit
S 34b014,4 hit
L 30a3fc,4 hit
S 34b018,4 hit
L 30a47c,4 hit
S 34b01c,4 hit
L 30a480,4 miss eviction
S 34a0a0,4 miss eviction
L 30a500,4 miss eviction
S 34a0a4,4 hit
L 30a580,4 miss eviction
S 34a0a8,4 hit
L 30a600,4 miss eviction
S 34a0ac,4 hit
L 30a680,4 miss eviction
S 34a0b0,4 hit
L 30a700,4 miss eviction
S 34a0b4,4 hit
L 30a780,4 miss eviction
S 34a0b8,4 hit
L 30a800,4 miss eviction
S 34a0bc,4 hit
L 30a484,4 hit
S 34a120,4 miss eviction
L 30a504,4 hit
S 34a124,4 hit
L 30a584,4 hit
S 34a128,4 hit
L 30a604,4 hit
S 34a12c,4 hit
L 30a684,4 hit
S 34a130,4 hit
L 30a704,4 hit
S 34a134,4 hit
L 30a784,4 hit
S 34a138,4 hit
```

```
S 34ad14,4 hit
L 30a3e4,4 hit
S 34ad18,4 hit
L 30a464,4 hit
S 34ad1c,4 hit
L 30a0e8,4 hit
S 34ad80,4 miss eviction
L 30a168,4 hit
S 34ad84,4 hit
L 30a1e8,4 hit
S 34ad88,4 hit
L 30a268,4 hit
S 34ad8c,4 hit
L 30a2e8,4 hit
S 34ad90,4 hit
L 30a368,4 hit
S 34ad94,4 hit
L 30a3e8,4 hit
S 34ad98,4 hit
L 30a468,4 hit
S 34ad9c,4 hit
L 30a0ec,4 hit
S 34ae00,4 miss eviction
L 30a16c,4 hit
S 34ae04,4 hit
L 30a1ec,4 hit
S 34ae08,4 hit
L 30a26c,4 hit
S 34ae0c,4 hit
L 30a2ec,4 hit
S 34ae10,4 hit
L 30a36c,4 hit
S 34ae14,4 hit
L 30a3ec,4 hit
S 34ae18,4 hit
L 30a46c,4 hit
S 34ae1c,4 hit
L 30a0f0,4 hit
S 34ae80,4 miss eviction
L 30a170,4 hit
S 34ae84,4 hit
L 30a1f0,4 hit
S 34ae88,4 hit
L 30a270,4 hit
S 34ae8c,4 hit
L 30a2f0,4 hit
S 34ae90,4 hit
L 30a370,4 hit
S 34ae94,4 hit
```

在缓存模拟信息中，大多数访存都如右图所示，这与先前的估计相符合。

但是每隔一段距离却会出现左图这样的集中 miss 出现的情况。

观察模拟缓存的输出文件可以推断出，A 和 B 的内存地址分别为

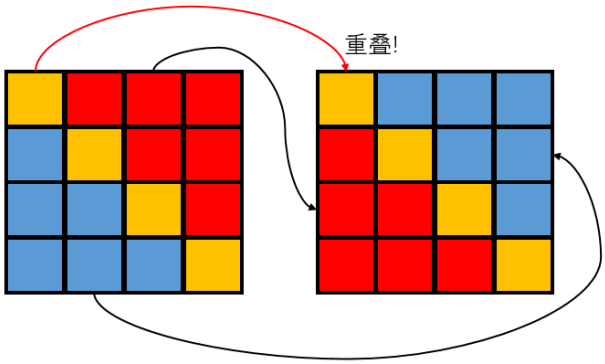
0x30a080 和 0x34a080

写出对应的二进制码：

A　0x30a080 : 00110100101000　　00100　　00000

B　0x34a080 : 00110000101000　　00100　　00000

发现对于矩阵 A 与 B 来说，相同的分块矩阵在内存地址上的组索引位是一样的。于是导致这样一个问题，非对角线上的分块矩阵的转置由于在两个矩阵中对应不同位置，所以缓冲区互不干扰，能够正常命中。而对角线上的四个分块矩阵由于转置后仍然存放在相同的对应分块矩阵上。



这种情况导致每次对对角线上的分块矩阵进行读写的时候，都会不断地产生发生冲突不命中。所以就会出现之前所看到的 miss 的情况。

因此，考虑对对角线上的四个分块矩阵特殊处理。

可以增加 8 个局部变量，一次性将一行的内容全部保存下来，然后一次性写入，这样就能够避免冲突不命中。代码如下：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i/*, j*/;
    int k, l;
    int val0, val1, val2, val3, val4, val5, val6, val7;

    // ...
    for (l = 0; l < 4; l++) {
        for (k = 0; k < 4; k++) {
            for (i = 8 * l; i < 8 *l + 8; i++) {
                val0 = A[i][0 + 8 * k];
                val1 = A[i][1 + 8 * k];
                val2 = A[i][2 + 8 * k];
                val3 = A[i][3 + 8 * k];
                val4 = A[i][4 + 8 * k];
                val5 = A[i][5 + 8 * k];
                val6 = A[i][6 + 8 * k];
                val7 = A[i][7 + 8 * k];
                B[0 + 8 * k][i] = val0;
                B[1 + 8 * k][i] = val1;
                B[2 + 8 * k][i] = val2;
                B[3 + 8 * k][i] = val3;
                B[4 + 8 * k][i] = val4;
                B[5 + 8 * k][i] = val5;
                B[6 + 8 * k][i] = val6;
                B[7 + 8 * k][i] = val7;
            }
        }
    }
}
```

测试结果

```
hit1170500820@1170500820:~/program-lab/HIT-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
hit1170500820@1170500820:~/program-lab/HIT-LAB6/cachelab-handout$
```

满分通过

64×64 程序设计思想

首先尝试直接分成 8x8 的分块矩阵进行测试，发现 miss 值非常高。

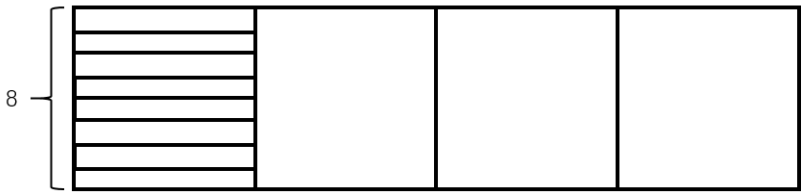
检查轨迹文件，得到 64x64 的矩阵 A B 的内存位置分别为：

0x30b0a0 和 0x34b0a0

且通过轨迹文件可以看到相当频繁的 miss 发生

```
S 34e1b0,4 hit
L 30b664,4 miss eviction
S 34e1b4,4 hit
L 30b764,4 miss eviction
S 34e1b8,4 hit
L 30b864,4 miss eviction
S 34e1bc,4 hit
L 30b168,4 miss eviction
S 34e2a0,4 miss eviction
L 30b268,4 miss eviction
S 34e2a4,4 hit
L 30b368,4 miss eviction
S 34e2a8,4 hit
L 30b468,4 miss eviction
S 34e2ac,4 hit
L 30b568,4 miss eviction
S 34e2b0,4 hit
L 30b668,4 miss eviction
S 34e2b4,4 hit
L 30b768,4 miss eviction
S 34e2b8,4 hit
L 30b868,4 miss eviction
S 34e2bc,4 hit
L 30b16c,4 miss eviction
```

原因很显然，cache 的配置为 32 组+每块 32 字节，对于 32x32 矩阵来说：



每块 32 字节存 8 个 int，总共能够存放 8 组，于是一个矩阵的一个 8x8 分块矩阵刚好能够无冲突的存放在 cache 里面。

而一个 64x64 的矩阵就会产生冲突：

退而求其次，尝试按照 4x4 的分块矩阵来分块求转置，

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;
    int k, l;

    for (l = 0; l < 16; l++) {
        for (k = 0; k < 16; k++) {
            for (j = 4 * k; j < 4 * k + 4; j++) {
                for (i = 4 * l; i < 4 * l + 4; i++) {
                    B[j][i] = A[i][j];
                }
            }
        }
    }
}
```

运行结果：

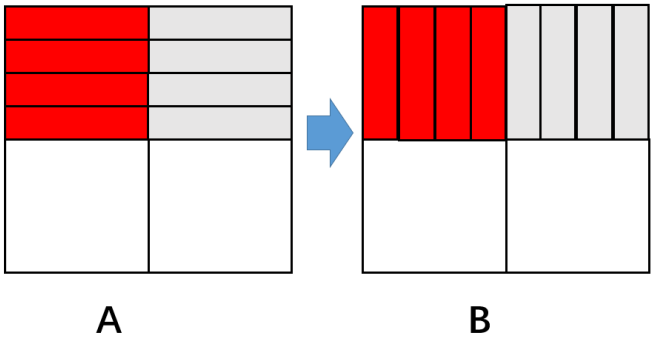
```
func 0 (Transpose submission): hits:6354, misses:1843, evictions:1811
```

可见虽然分块更小，对 cache 的利用度更小，但是由于没有冲突不命中，miss 次数反而大大减小。但是 1843 任然大于 1300，说明要想 miss 值低于 1300，就需要对 cache 区域进行更加充分的利用。

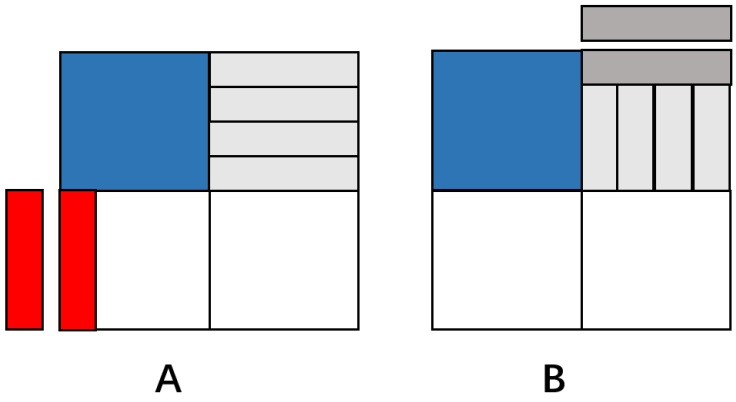
得到高人指点，尝试分块先把 A 中的数据放入 B 中，再对 B 进行处理。

基本思想如下：

首先将 A 的上侧 4x8 的元素转置后移到 B 中，由于一个缓冲块能够存储 8 个 int 类型的值，所以再访问 B 中第一列四个元素时，B 的上 4x8 的块就都已经在 cache 里了。这一步节省了一定的 cache 空间。



然后再将 A 的左下角的元素逐步移到 B 里面。这一步的顺序很重要，因为要充分利用 cache 的空间，所以需要先将 A 左下角分块矩阵的第一列和 B 的右上角分块矩阵的第一行用临时变量存起来



需要注意，B 矩阵的第一行和第五行会分配到 cache 的同一组而产生冲突

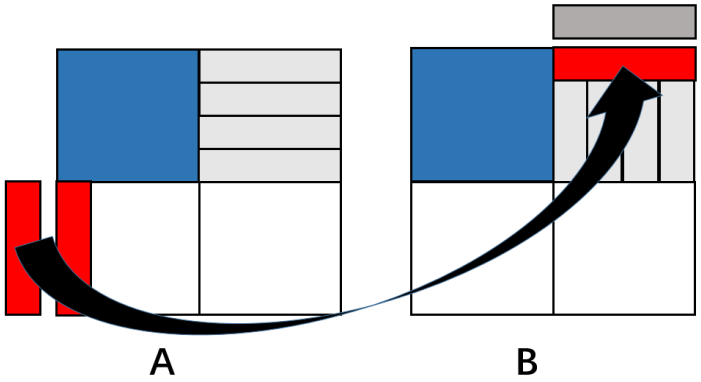
因此在这里，如果先将 B 右上分块矩阵的第一行移到左下的第一行，就会驱逐 B 中第一行存在缓冲区里的数据，而将 B 的第五行，也就是下半分块矩阵的第一行写入缓冲区。然后再将 A 中的元素写入 B 右上分块矩阵的话，又会产生一次驱逐。这就是两次 miss。如果这样操作的话，最后的测试结果：

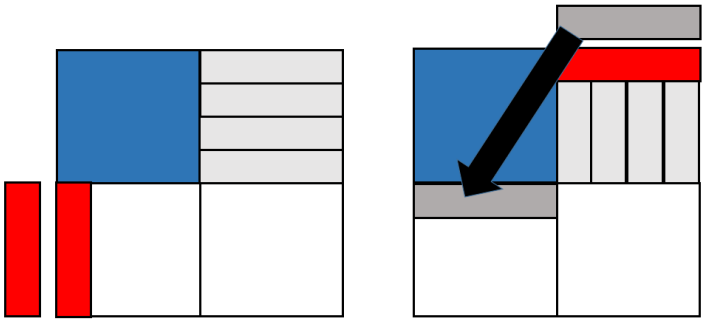
```
func 0 (Transpose submission): hits:8578, misses:1667, evictions:1635
```

1667 次显然时不合格的。

但是如果换一个顺序：

先将 A 左下分块矩阵的数据写入 B 右上分块矩阵，再将 B 右上分块矩阵的值写入 B 的右下分块矩阵，由于 A 在写入时已经将 B 的第一行放入了 cache，接下来 B 再写入的时候就能够直接从 cache 里取数据了。这样就减少了一半的 miss。





其余行同理，最终完成 B 中除右下角之外其它三个矩阵的移动和转置

B 右下角的分块矩阵只需简单的用临时变量按行转置后移入即可。

完整实现代码如下：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i;
    int k, l;
    int temp0, temp1, temp2, temp3, temp4, temp5, temp6, temp7;
    for (k = 0; k < 8; k++) {
        for (l = 0; l < 8; l++) {
            for (i = 0; i < 4; i++) {//第一个4x8的长方形转化赋值到B中,前4x4的正方形先转置
                temp0 = A[8 * l + i][8 * k + 0];
                temp1 = A[8 * l + i][8 * k + 1];
                temp2 = A[8 * l + i][8 * k + 2];
                temp3 = A[8 * l + i][8 * k + 3];
                temp4 = A[8 * l + i][8 * k + 4];
                temp5 = A[8 * l + i][8 * k + 5];
                temp6 = A[8 * l + i][8 * k + 6];
                temp7 = A[8 * l + i][8 * k + 7];
                B[8 * k + 0][8 * l + i] = temp0;
                B[8 * k + 1][8 * l + i] = temp1;
                B[8 * k + 2][8 * l + i] = temp2;
                B[8 * k + 3][8 * l + i] = temp3;
                B[8 * k + 0][8 * l + i + 4] = temp4;
                B[8 * k + 1][8 * l + i + 4] = temp5;
                B[8 * k + 2][8 * l + i + 4] = temp6;
                B[8 * k + 3][8 * l + i + 4] = temp7;
            }
            for (i = 0; i < 4; i++) {//A左下角与B右上角置换
                temp0 = A[8 * l + 4][8 * k + i];
                temp1 = A[8 * l + 5][8 * k + i];
                temp2 = A[8 * l + 6][8 * k + i];
                temp3 = A[8 * l + 7][8 * k + i];
                temp4 = B[8 * k + i][8 * l + 4];
                temp5 = B[8 * k + i][8 * l + 5];
```

```
                temp6 = B[8 * k + i][8 * l + 6];
                temp7 = B[8 * k + i][8 * l + 7];
                B[8 * k + i][8 * l + 4] = temp0;
                B[8 * k + i][8 * l + 5] = temp1;
                B[8 * k + i][8 * l + 6] = temp2;
                B[8 * k + i][8 * l + 7] = temp3;
                B[8 * k + i + 4][8 * l + 0] = temp4;
                B[8 * k + i + 4][8 * l + 1] = temp5;
                B[8 * k + i + 4][8 * l + 2] = temp6;
                B[8 * k + i + 4][8 * l + 3] = temp7;
            }
            for (i = 0; i < 4; i++) {//AB右下角置换，同时转置k
                temp0 = A[8 * l + i + 4][8 * k + 4];
                temp1 = A[8 * l + i + 4][8 * k + 5];
                temp2 = A[8 * l + i + 4][8 * k + 6];
                temp3 = A[8 * l + i + 4][8 * k + 7];
                B[8 * k + 4][8 * l + i + 4] = temp0;
                B[8 * k + 5][8 * l + i + 4] = temp1;
                B[8 * k + 6][8 * l + i + 4] = temp2;
                B[8 * k + 7][8 * l + i + 4] = temp3;
            }
        }
    }
}
```

最终测试结果：

```
hit1170500820@1170500820:~/program-lab/HIT-LAB6/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf hit1170500820-handin.tar  csim.c trans.c
csim.c
trans.c
hit1170500820@1170500820:~/program-lab/HIT-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:9066, misses:1179, evictions:1147

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=1179

TEST_TRANS_RESULTS=1:1179
```

61×67 程序设计思想

首先尝试 4x4 的分块方法，这种分块方法至少能够保证不会出现冲突不命中。

同时也采用临时变量存储的优化方法：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;
    int k, l;
    int temp0, temp1, temp2, temp3;

    for (l = 0; l < 16; l++) {
        for (k = 0; k < 15; k++) {
            for (j = 4 * k; j < 4 * k + 4; j++) {
                //for (i = 4 * l; i < 4 * l + 4; i++) {
                    //B[j][i] = A[i][j];
                    temp0 = A[4 * l][j];
                    temp1 = A[4 * l + 1][j];
                    temp2 = A[4 * l + 2][j];
                    temp3 = A[4 * l + 3][j];
                    B[j][4 * l] = temp0;
                    B[j][4 * l + 1] = temp1;
                    B[j][4 * l + 2] = temp2;
                    B[j][4 * l + 3] = temp3;
                //}
            }
        }
    }
    for (j = 0; j < 64; j++)
    {
        B[60][j] = A[j][60];
    }
    for (j = 64; j < 67; j++) {
        for (i = 0; i < 61; i++) {
            B[i][j] = A[j][i];
        }
    }
```

产生结果如下：

```
func 0 (Transpose submission): hits:5921, misses:2258, evictions:2226
```

因此这种方法仍然是对 cache 利用不够。

考虑到 61>60，改为 15x15 试试(未加临时变量)：

```
func 0 (Transpose submission): hits:5917, misses:2262, evictions:2230
```

再改为 15x16 试试

```
func 0 (Transpose submission): hits:6078, misses:2101, evictions:2069
```

已经相当接近了。

添加了临时变量后：

```
func 0 (Transpose submission): hits:6146, misses:2033, evictions:2001
```

已经无法再优化了，只能改框架，尝试采用 16x16 与临时变量：

```
hit1170500820@1170500820:~/program-lab/HIT-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:6216, misses:1963, evictions:1931

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=1963

TEST_TRANS_RESULTS=1:1963
```

通过

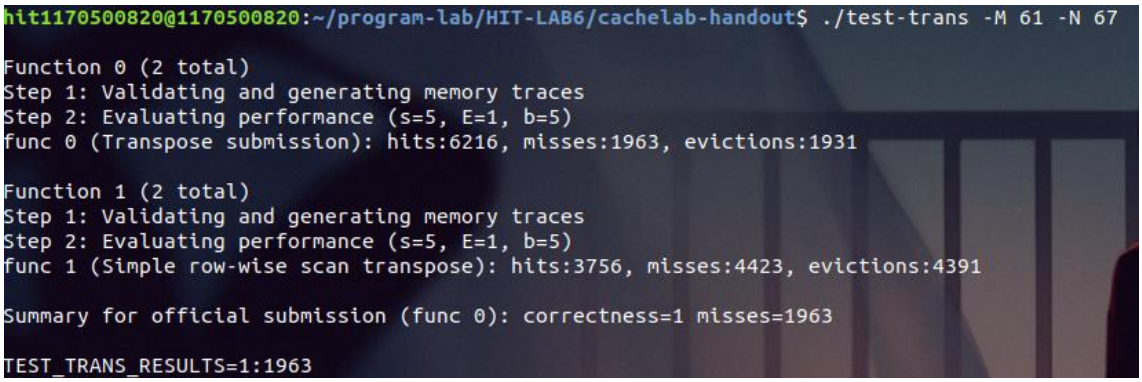
32×32（10 分）：运行结果截图

```
hit1170500820@1170500820:~/program-lab/HIT-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
hit1170500820@1170500820:~/program-lab/HIT-LAB6/cachelab-handout$
```

64×64（10 分）：运行结果截图

```
hit1170500820@1170500820:~/program-lab/HIT-LAB6/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf hit1170500820-handin.tar  csim.c trans.c
csim.c
trans.c
hit1170500820@1170500820:~/program-lab/HIT-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:9066, misses:1179, evictions:1147

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=1179

TEST_TRANS_RESULTS=1:1179
```

61×67（20 分）：运行结果截图



第 4 章 总结

4.1 请总结本次实验的收获

本次实验中我对 cache 的基本工作原理有了相当全面而具体的认识。通过编写冗长无味而 bug 百出的模拟 cache 运行的 csim 程序，亲身实践来实现一个简单的 cache。其中 lru 的优化效果非常具体的呈现在眼前，一点小小的改动都能带来缓存命中次数不小的提升，这使我意识到计算机编程是一门很深厚的学问。

4.2 请给出对本次实验内容的建议

我认为实验的教程 ppt 不能够很好的指导我们去做这个实验。
ppt 中对某些工具的使用，对某些文件的作用并没有解释的很清楚，对于实验的具体操作没能给出很好的指示。
比如说 cmu 给出的实验指导 ppt 中对矩阵转置实验就给出了“注意对角线”这样的提示，这种一语中的的精妙的提示胜过冗长的手把手教。
如果不是听说了 cmu 教程中有这样一句话，我花在这个实验上的时间恐怕又要增加不少。

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.