

哈尔滨工业大学

实验报告

实 验（六）

题 目 Cachelab

高速缓冲器模拟

专 业 计算机

学 号 1180300308

班 级 03003

学 生 刘义

指 导 教 师 史先俊

实 验 地 点 管理楼 712

实 验 日 期 2019 年 11 月 27 日

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	- 4 -
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）	- 4 -
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）	- 4 -
2.3 写出各类 CACHE 的读策略与写策略（5 分）	- 4 -
2.4 写出用 GPROF 进行性能分析的方法（5 分）	- 4 -
2.5 写出用 VALGRIND 进行性能分析的方法（（5 分）	- 4 -
第 3 章 CACHE 模拟与测试.....	- 5 -
3.1 CACHE 模拟器设计	- 5 -
3.2 矩阵转置设计	- 5 -
第 4 章 总结	- 6 -
4.1 请总结本次实验的收获	- 6 -
4.2 请给出对本次实验内容的建议	- 6 -
参考文献	- 7 -

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

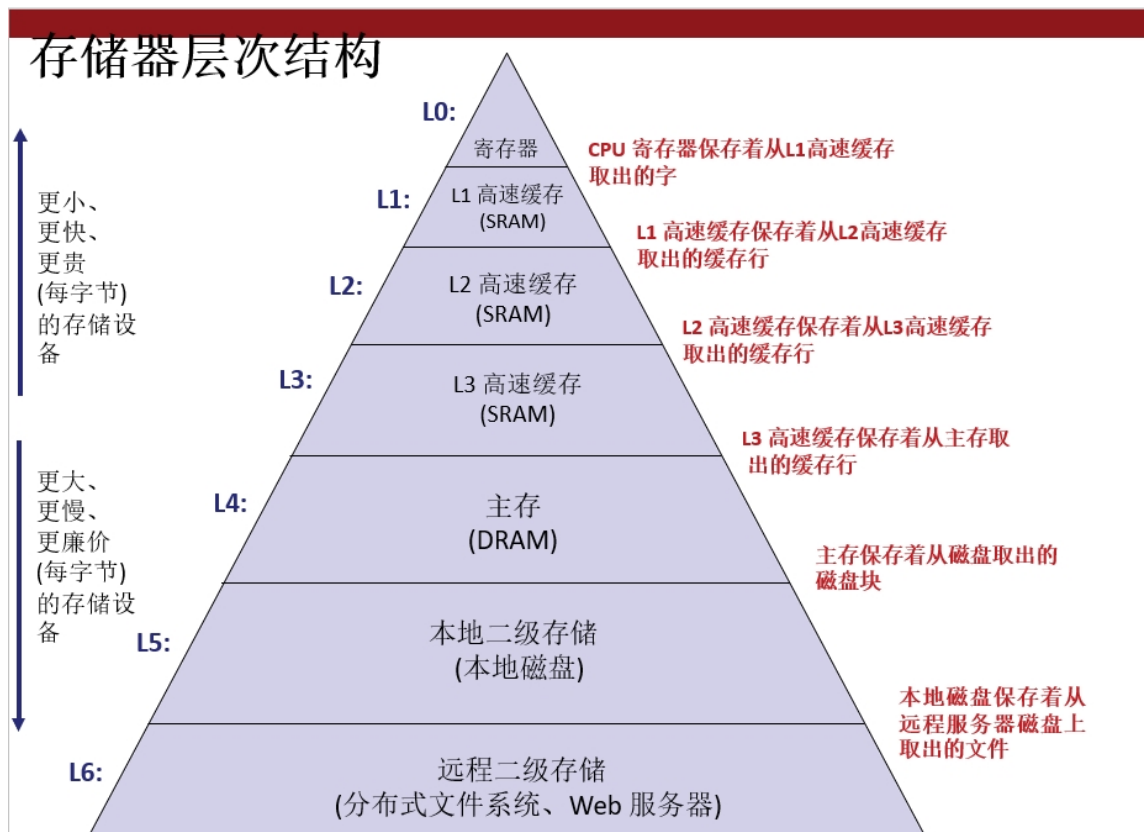
1.3 实验预习

- 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 C S E B c s e b

- 写出 Cache 的基本结构与参数
- 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)

CPU-Z

处理器 | 缓存 | 主板 | 内存 | SPD | 显卡 | 测试分数 | 关于

处理器

名字	Intel Core i7 8550U				
代号	Kaby Lake-U/Y	TDP	15.0 W		
插槽	Socket 1356 FCBGA				
工艺	14 纳米	核心电压	1.262 V		
规格	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz				
系列	6	型号	E	步进	A
扩展系列	6	扩展型号	8E	修订	U0
指令集	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3				

时钟 (核心 #0)

核心速度	3690.97 MHz
倍频	x 37.0 (4 - 40)
总线速度	99.76 MHz
额定 FSB	

缓存

一级 数据	4 x 32 KBytes	8-way
一级 指令	4 x 32 KBytes	8-way
二级	4 x 256 KBytes	4-way
三级	8 MBytes	16-way

已选择 处理器 #1 核心数 4 线程数 8

CPU-Z Ver. 1.90.0.x64 工具 验证 确定

CPU-Z

处理器 | 缓存 | 主板 | 内存 | SPD | 显卡 | 测试分数 | 关于

一级数据缓存

大小	32 KBytes	x 4
描述	8-way set associative, 64-byte line size	

一级指令缓存

大小	32 KBytes	x 4
描述	8-way set associative, 64-byte line size	

二级缓存

大小	256 KBytes	x 4
描述	4-way set associative, 64-byte line size	

三级缓存

大小	8 MBytes	
描述	16-way set associative, 64-byte line size	

大小 描述 速度

CPU-Z Ver. 1.90.0.x64 工具 验证 确定

	C	S	E	B	s	e	b
L1	8*32KB	64	8	64	6	3	6
L2	4*256KB	2^{10}	4	64	10	2	6
L3	8MB	2^{13}	16	64	13	4	6

2.3 写出各类 Cache 的读策略与写策略 (5 分)

读写数据需要传递一个 `addr`，`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，最后通过块偏移定位需要的数据在行内的位置。

读策略：如果命中返回，否则从主存或下一级 `cache` 中读取数据，并替换出一行数据，通常采用 LRU 算法进行驱逐和替换。

写策略：若命中，分两种策略：

1) 写回法：只写本级 `cache`，暂时不写数据到主存或下一级 `cache`，等到该行被驱逐时，才将数据写回到主存或下一级 `cache`。

2) 直写法：写本级 `cache`，同时写数据到主存或下一级 `cache`，等到该行被驱逐时，就不用写回数据了。

若不命中，也分两种策略：

1) 写分配，又分两种：[1]先写数据到主存或下一级 `cache`，并从主存或下一级 `cache` 读取刚才修改过的数据，即：先写数据，再为所写数据分配 `cache line`；[2]先分配 `cache line` 给所写数据，即：从主存中读取一行数据到 `cache`，然后直接对 `cache` 进行修改，并不把数据写到主存或下一级 `cache`，一直等到该行被替换出去，才写数据到主存或下一级 `cache`。

2) 写不分配：直接写数据到主存或下一级 `cache`，并且不从主存或下一级 `cache` 中读取被改写的的数据，即：不分配 `cache line` 给被修改的数据。

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。

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 github 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 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 `L1` 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](http://www.valgrind.org)
- 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 是运行程序的进程 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：

首先，Cache 组数 $S=2^s$ ， s 为组索引位数；块字节数 $B=2^b$ ， b 为块地址位数。

然后，来看 Cache 初始化函数：initCache

Cache 有 S 个组，每个组有 E 行，每一行都被声明为一个结构体，因此可用二维数组表示：

```
void initCache()
{
    cache = (cache_t)malloc(sizeof(cache_set_t) * S);
    for (int i = 0; i < S; i++) {
        cache[i] = (cache_set_t)malloc(sizeof(cache_line_t) * E);
        for (int j = 0; j < E; j++) {
            cache[i][j].lru = 0; //初始化访问标记
            cache[i][j].tag = 0; //初始化标记位
            cache[i][j].valid = 0; //初始化有效位
        }
    }
}
```

Cache 释放函数 freeCache 很简单，反过来释放即可：

```
/*
 * freeCache - free allocated memory
 */
void freeCache()
{
    for (int i = 0; i < S; i++) {
        free(cache[i]);
    }
    free(cache);
}
```

最后，我们来看函数 `accessData`

1. 通过传进函数的地址计算出标记位 CT，组索引 CI，位偏移 CO；
2. 到组 CI 中去寻找标记位为 CT 且有效位为 1 的行，若找到，（输出 `hit`，）命中次数加 1，更改访问标记，函数返回；否则（输出 `miss`，）不命中次数加 1，然后见 3
3. 在组 CI 中寻找有效位为 0 的行，若找到，将其有效位置 1，标记位置 CT，更改访问标记，函数返回；否则，（输出 `eviction`，）驱逐次数加 1，然后见 4
4. 在组 CI 中寻找访问标记最小的行，将其标记位置 CT，更改访问标记，函数返回。
5. 更改访问标记：让 当前访问行的访问标记 = 该组所有行的访问标记的最大值+1

函数截图：

```

{
    int CT = addr / (S*B);
    int CI = (addr % (S*B)) / B;
    int CO = (addr % (S*B)) % B;
    unsigned long long int LRU = cache[CI][0].lru;
    for (int i = 1; i < E; i++) {
        if (cache[CI][i].lru > LRU) {
            LRU = cache[CI][i].lru;
        }
    }
    for (int i = 0; i < E; i++) {
        if (cache[CI][i].valid == 1 && cache[CI][i].tag == CT) {
            if (verbosity) {
                printf("hit ");
            }
            hit_count++;
            cache[CI][i].lru = LRU + 1;
            return;
        }
    }
    if (verbosity) {
        printf("miss ");
    }
    miss_count++;
    for (int i = 0; i < E; i++) {
        if (cache[CI][i].valid == 0) {
            cache[CI][i].tag = CT;
            cache[CI][i].valid = 1;
            cache[CI][i].lru = LRU + 1;
            return;
        }
    }
    if (verbosity) {
        printf("eviction ");
    }
    eviction_count++;
    int Flag = 0;
    for (int i = 1; i < E; i++) {
        if (cache[CI][i].lru < cache[CI][Flag].lru) {
            Flag = i;
        }
    }
    cache[CI][Flag].tag = CT;
    cache[CI][Flag].lru = LRU + 1;
}

```

测试用例 1 的输出截图（5 分）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/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）：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/cachelab-handout$ ./csim -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

3.2 矩阵转置设计

提交 trans.c

程序设计思想:

测试 Cache 有 32 个组，每个组有一行，每一行（块）有 32 字节。矩阵元素为 int 类型，占 4 个字节，则 cache 每个块包含 8 个矩阵元素。

1) 32×32

对 32×32 矩阵，Cache 读入连续的 8 行不会引发驱逐，而第 9 行会驱逐第 1 行的相应元素。因此采用 8×8 矩阵的方式去读写，即将 32×32 矩阵划分成若干个 8×8 矩阵，然后逐个矩阵进行转置。

此时 miss 次数为 343 次，注意到当读写对角线上的元素时，写矩阵 B 时会将矩阵 A 中对应块驱逐掉，然后读矩阵 A 时又会将矩阵 B 中对应块驱逐掉。为进一步降低 miss 次数，将 32×32 矩阵中对角线上的元素读写放到该元素所在 8×8 矩阵所在行的最后，这样 miss 次数降低到 287。

代码为:

```
if (N == 32 || N == 67) {
    for (y = 0; y < M; y = y + 8) {
        for (x = 0; x < N; x = x + 8) {
            for (i = 0; i < 8 && x + i < N; i++) {
                for (j = 0; j < 8 && y + j < M; j++) {
                    if (x + i != y + j) {
                        tmp = A[x + i][y + j];
                        B[y + j][x + i] = tmp;
                    }
                }
                if (x == y) {
                    tmp = A[x + i][x + i];
                    B[x + i][x + i] = tmp;
                }
            }
        }
    }
}
```

3) 61×67 : 同 32×32

2) 64×64

对 64×64 矩阵，Cache 读入连续的 4 行不会引发驱逐，而第 5 行会驱逐第 1 行的相应元素。若采用 4×4 矩阵的方式去读写，miss 大约 1800 次。

为进一步降低 miss 次数，采用 8×8 矩阵的方式去读写，每个 8×8 矩阵又分为 4 个 4×4 矩阵。对每个 8×8 矩阵：

1) 首先将矩阵 A 左上方的 4×4 矩阵转置

2) 再将矩阵 A 右上方的 4×4 矩阵放到矩阵 B 的右上方

3) 然后先将矩阵 B 右上方 4×4 矩阵的一行 4 个元素放到寄存器变量中，再将矩阵 A 左下方 4×4 矩阵的一行 4 个元素放到矩阵 B 右上方 4×4 矩阵的那一行中，最后将 4 个寄存器变量放到矩阵 A 左下方 4×4 矩阵的那一行中。重复进行 4 次，完成矩阵 A 左下方和右上方的 4×4 矩阵的转置。

4) 最后将矩阵 A 右下方的 4×4 矩阵转置

代码如下：


```

41  if (N == 64) {
42      for (x = 0; x < N; x = x + 8) {
43          for (y = 0; y < M; y = y + 8) {
44              if (x == y) {
45                  for (i = 0; i < 4; i++) {
46                      for (j = 0; j < 4; j++) {
47                          if (x + i != y + j) {
48                              tmp = A[x + i][y + j];
49                              B[y + j][x + i] = tmp;
50                          }
51                      }
52                      tmp = A[x + i][x + i];
53                      B[x + i][x + i] = tmp;
54                  }
55                  for (; i < 8; i++) {
56                      for (j = 0; j < 4; j++) {
57                          tmp = A[x + i][y + j];
58                          B[y + j][x + i] = tmp;
59                      }
60                  }
61                  for (i = 4; i < 8; i++) {
62                      for (j = 4; j < 8; j++) {
63                          if (x + i != y + j) {
64                              tmp = A[x + i][y + j];
65                              B[y + j][x + i] = tmp;
66                          }
67                      }
68                      tmp = A[x + i][x + i];
69                      B[x + i][x + i] = tmp;
70                  }
71                  for (i = 0; i < 4; i++) {
72                      for (j = 4; j < 8; j++) {
73                          tmp = A[x + i][y + j];
74                          B[y + j][x + i] = tmp;
75                      }
76                  }
77              }
78          }
79      }
80      else {
81          for (i = 0; i < 4; i++) {
82              for (j = 0; j < 4; j++) {
83                  tmp = A[x + i][y + j];
84                  B[y + j][x + i] = tmp;
85              }
86          }
87          for (; j < 8; j++) {
88              tmp = A[x + i][y + j];
89              B[y + j - 4][x + i + 4] = tmp;
90          }
91          for (j = 0; j < 4; j++) {
92              tmp = B[y + j][x + 4];
93              tmp1 = B[y + j][x + 5];
94              tmp2 = B[y + j][x + 6];
95              tmp3 = B[y + j][x + 7];
96              tmp4 = A[x + 4][y + j];
97              tmp5 = A[x + 5][y + j];
98              tmp6 = A[x + 6][y + j];
99              tmp7 = A[x + 7][y + j];
100             B[y + j][x + 4] = tmp4;
101             B[y + j][x + 5] = tmp5;
102             B[y + j][x + 6] = tmp6;
103             B[y + j][x + 7] = tmp7;
104             B[y + j + 4][x] = tmp;
105             B[y + j + 4][x + 1] = tmp1;
106             B[y + j + 4][x + 2] = tmp2;
107             B[y + j + 4][x + 3] = tmp3;
108         }
109         for (i = 4; i < 8; i++) {
110             for (j = 4; j < 8; j++) {
111                 tmp = A[x + i][y + j];
112                 B[y + j][x + i] = tmp;
113             }
114         }
115     }
116 }
117 }
118

```

32×32 (10 分) : 运行结果截图

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/cachelab-handout$ ./test-trans -M 32 -N 32

Function 0 (4 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
```

64×64 (10 分) : 运行结果截图

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/cachelab-handout$ ./test-trans -M 64 -N 64

Function 0 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:8690, misses:1299, evictions:1267
```

61×67 (20 分) : 运行结果截图

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab6/cachelab-handout$ ./test-trans -M 61 -N 67

Function 0 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6268, misses:1917, evictions:1885
```

第 4 章 总结

4.1 请总结本次实验的收获

- 1) 深入了解了 Cache 的工作机制
- 2) 尝试编写 Cache 友好的代码

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>.