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 1170300817

班　　级 1703008

学　　生 林之浩

指 导 教 师 郑贵滨

实 验 地 点 G712

实 验 日 期 11.19

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

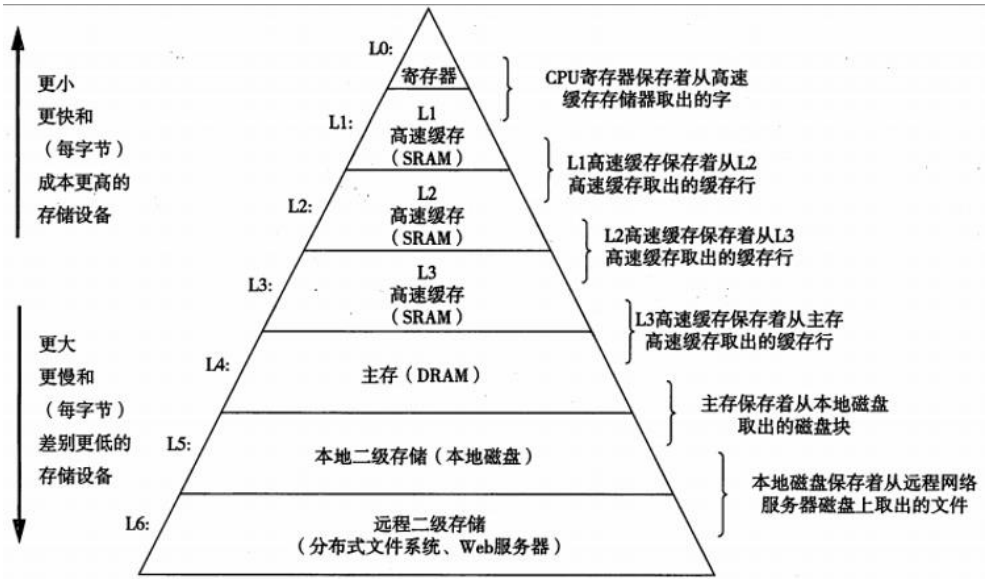
Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind

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
写出 Cache 的基本结构与参数
写出各类 Cache 的读策略与写策略

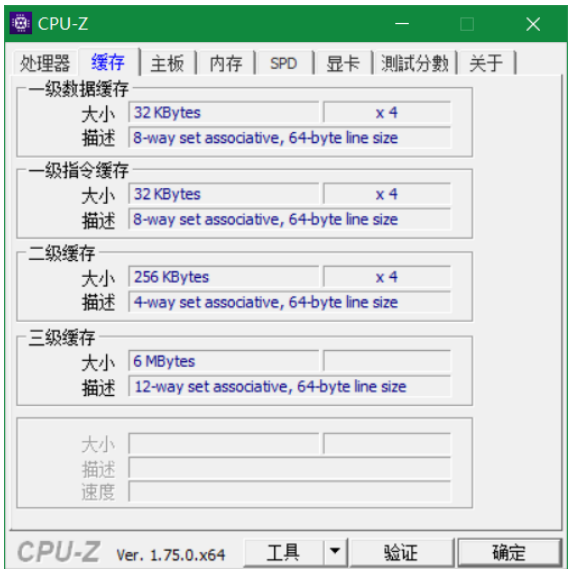
第 2 章 实验预习

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 数据	4x32 KBytes	64	8	64	6	3	6
L1 指令	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：

当计算机要从内存中读取一个值的时候，cpu 会先到最高一级的 L1 cache 中去寻找是否有该内存块，如果有，这就是我们说的缓存命中，该程序直接从该处读取数据。

若没有，计算机就会到下一个层级的 cache 中去寻值，依次类推，直到找到为止。然后一层一层往高层传递，期间会发成驱逐等情况。

写策略：

写策略有两种，

第一种叫做直写，就是立即将某个数据块直接写回到紧接着的低一层中去。虽然快，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。能显著减少总线流量。

而在处理写不命中时，也有两种方法

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）

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

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：

```
void initCache();
void freeCache();
int getSet(mem_addr_t addr);
int getTag(mem_addr_t addr);
int Miss(mem_addr_t addr);
void updateLru(int hitIndex);
int findLru();
int updateCache(mem_addr_t addr);
void accessData(mem_addr_t addr);
```

getSet 获取 set 值，getTag 获取 tag 值，并且我们每次只处理一条数据所以 set 和 tag 值设成了全局变量所以传的参数很少。

```
void accessData(mem_addr_t addr)
{
    if(Miss(addr)==1)
    {
        miss_count++;
        if(verbosity == 1)
            printf("miss ");
        if(updateCache(addr) == 1)
        {
            eviction_count++;
            if(verbosity==1)
                printf("eviction ");
        }
    }
    else
    {
        hit_count++;
        if(verbosity == 1)
            printf("hit ");
    }
}
```

主要工作就是完成这个访问数据的函数，读写就执行一次，替换就相当于一读一写，执行两次。封装完成后这个函数看起来就十分得简洁了，无非 miss 之后 miss_count++，然后更新 cache，驱逐次数++，如果命中就命中次数++。下面看具体函数。

```
int Miss(mem_addr_t addr)
{
    int j;
    int Miss = 1;
    for(j=0; j<E; j++)
    {
        if(cache[SET][j].valid == 1 && cache[SET][j].tag == TAG)
        {
            Miss = 0;
            updateLru(j);
        }
    }
    return Miss;
}
```

Miss 函数很简单，根据计算的 SET 值和 TAG 值到我们虚拟的 cache 里（其实是一个大二维结构体数组）找一找，找到了就没有 miss 更新 lru 值（因为这个数据被访问了一次），否则返回 miss

```
void updateLru(int hitIndex)
{
    cache[SET][hitIndex].lru = 99999;
    int j;
    for(j=0; j<E; j++)
    {
        cache[SET][j].lru--;
    }
}
```

Lru 更新时我们把传入的要更新的块的 lru 值设置成 99999，也就是很大，然后对所有块的 lru 数字减一，这样，这个块就是 lru 最大的数据块了（最近访问）

```
int updateCache(mem_addr_t addr)
{
    int j;
    int isfull = 1;
    for(j=0; j<E; j++)
    {
        if(cache[SET][j].valid== 0)
        {
            isfull = 0;
            break;
        }
    }
    if(isfull == 0)
    {
        cache[SET][j].valid= 1;
        cache[SET][j].tag = TAG;
        updateLru(j);
    }
    else
    {
        int evicIndex = findLru(SET);
        cache[SET][evicIndex].valid = 1;
        cache[SET][evicIndex].tag = TAG;
        updateLru(evicIndex);
    }
    return isfull;
}
```

更新 cache 时要先判断是否已满，如果未满就在不满的地方放置，已满就要找到 lru 数字最小的数据块进行驱逐，findlru 函数就是遍历找到最小 lru 数字的函数很简单就不解释了。找到之后将其驱逐，无论哪种结果最后都要更新 lru 数值。

测试用例 1 的输出截图（5 分）：

测试用例 2 的输出截图（5 分）：

测试用例 3 的输出截图（5 分）：

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）：

测试用例 7 的输出截图（5 分）：

测试用例 8 的输出截图（10 分）：

```
linleo@ubuntu:~/Desktop/cachelab-handout$ ./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
```

3.2 矩阵转置设计

提交 trans.c

```
linleo@ubuntu:~/Desktop/cachelab-handout$ ./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        1992
          Total points    53.0        53
linleo@ubuntu:~/Desktop/cachelab-handout$
```

程序设计思想：因为实验只测试三个特定的参数，于是我们只要针对这三个特定的参数分别进行优化即可。

32×32（10 分）：

```
linleo@ubuntu:~/Desktop/cachelab-handout$ ./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
```

首先计算 cache 的各项参数（s=5、E=1、b=5），所以 S=32,B=32 字节,int 四个字节，所以一个 B 能存 8 个 int 数据，整个 cache 一共能存 32*8 个数据，对应实际矩阵的 8 个行，也就是说，相隔 8 行的元素在读到 cache 的时候是映射到同一块内存中的，比如我们先后读取第 1 行和第 9 行的前几个元素，就会造成很多替换。这是我们不想见到的。

所以我们将整个矩阵划分成 8*8 的方块来分别处理。

此外，因为整个矩阵的大小是 cache 的整数倍，所以 A,B 中下标相同的元素也有很大的造成冲突的可能性，比如我们的访问顺序是 A[0][0]->B[0][0]->A[0][1],当我们访问 B[0][0]的时候，cache 中的 A[0][0]—A[0][8]整个块会被驱逐，这样当我们访问 A[0][1]时又会出现新的冲突不命中，这样的情况对于每个对角线元素都是存在的。我们要力求避免。

于是考虑用寄存器来实现，每当我们处理位于对角线上的 8*8 方块时，我们就用 8 个寄存器一次把块里面的 8 个数都读出来，这样后续不用再访问 cache 请求这些数据，就不会在读取 B 矩阵对应元素之后出现冲突不命中了。

对于不在对角线上的 8*8 的方块，只需要遍历后 B[y][x]= A[x][y];即可

代码如下

```
{
    int x0,x1,x2,x3,x4,x5,x6,x7;
    if (N == 32)
    {
        for (int i = 0; i < N; i += 8)
        {
            for (int j = 0; j < M; j += 8)
            {
                for (int x = i; x < i + 8; ++x)
                {
                    if (i==j)
                    {
                        x0 = A[x][j];
                        x1 = A[x][j+1];
                        x2 = A[x][j+2];
                        x3 = A[x][j+3];
                        x4 = A[x][j+4];
                        x5 = A[x][j+5];
                        x6 = A[x][j+6];
                        x7 = A[x][j+7];
                        B[j][x] = x0;
                        B[j+1][x] = x1;
                        B[j+2][x] = x2;
                        B[j+3][x] = x3;
                        B[j+4][x] = x4;
                        B[j+5][x] = x5;
                        B[j+6][x] = x6;
                        B[j+7][x] = x7;
                    }
                    else
                    {
                        for (int y = j; y< j + 8; ++y)
                        {
                            B[y][x]= A[x][y];
                        }
                    }
                }
            }
```

64×64（10 分）：

这时候矩阵一行有 64 个元素了，4 行矩阵就能填满整个 cache，所以我们访问相隔 4 行的元素就会造成冲突不命中。

自然想到按第一题的思路划分成 4*4

```
else if (N == 64)
{
    for (int i = 0; i < N; i += 4)
    {
        for (int j = 0; j < M; j += 4)
        {
            for (int x = i; x < i + 4; ++x)
            {
                if (i==j)
                {
                    x0 = A[x][j];
                    x1 = A[x][j+1];
                    x2 = A[x][j+2];
                    x3 = A[x][j+3];

                    B[j][x] = x0;
                    B[j+1][x] = x1;
                    B[j+2][x] = x2;
                    B[j+3][x] = x3;

                }
                else
                {
                    for (int y = j; y< j + 4; ++y)
                    {
                        B[y][x]= A[x][y];
                    }
                }
            }
        }
    }
```

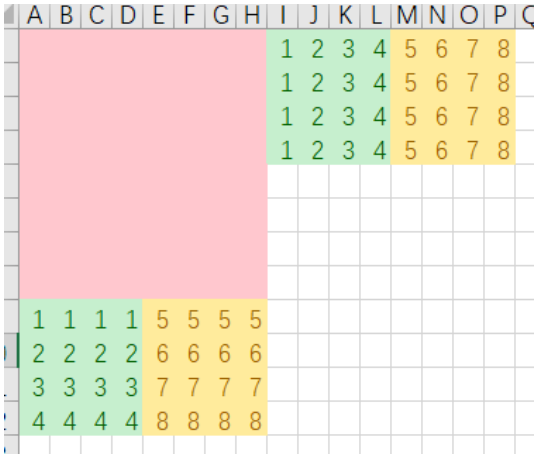
但是

```
linleo@ubuntu:~/Desktop/cachelab-handout$ ./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:6402, misses:1795, evictions:176

Summary for official submission (func 0): correctness=1 misses=1795

TEST_TRANS_RESULTS=1:1795
```

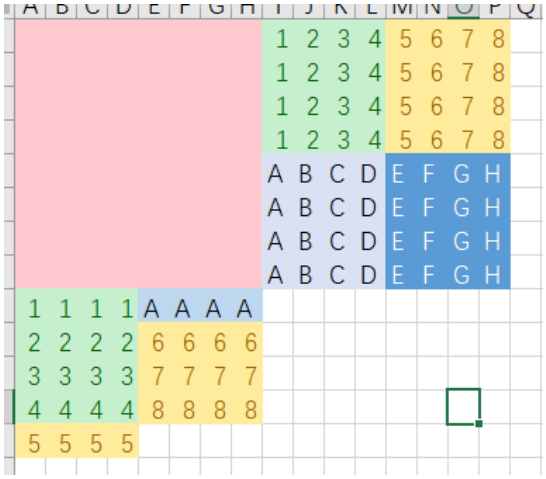
效果不好没达到要求的 1300.

究其原因还是因为一次读取 8 个数但是只用上了 4 个，利*用率太低

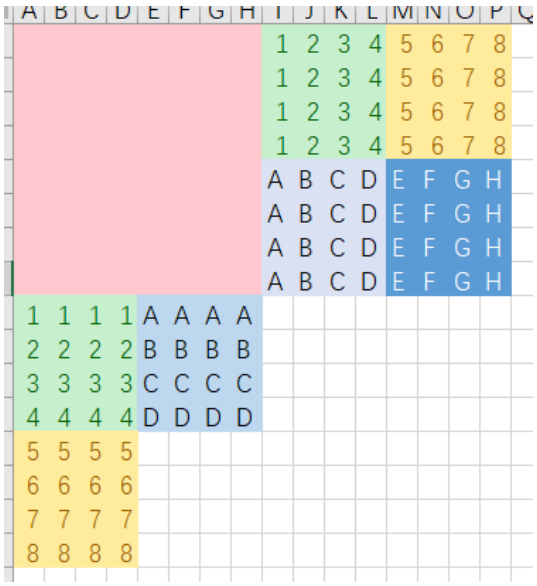
为了避免一次访问 B 矩阵的时候跨越 8 行，对于每一个 8*8 小矩阵我们采取先移后转置的策略，1、先读取前四行 4*8 的 A 矩阵，转置成两个 4*4 并排码放在 B 中，此时第二个 4*4 矩阵的位置还不正确，但已经转置。

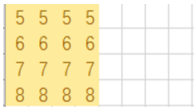


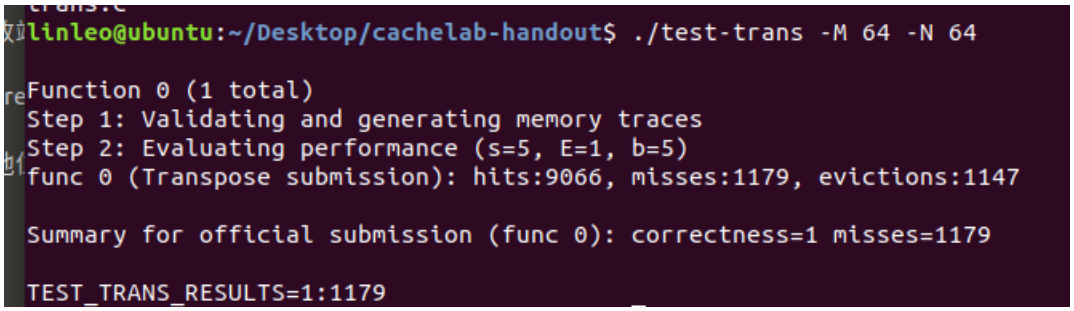
随后读取左下的 4*4 矩阵，这时我们就能按列读取 4*4 矩阵了，然后按行放置于刚才的错误位置矩阵处并将错误矩阵归位。



放置一排 5 的时候会在 cache 中驱逐 1111 那一行但是此时 AAAA 已经放置成功所以我们没有浪费 cache. 以下类似。



到了这里我们只需移动最后的 4*4 方块，而且这时存在 cache 里的既有  又有  ，十分方便。不会出现 miss



61×67（20 分）

对于实验的最后一部分，其实要求很低，miss 数量小于 2000 就可以了

```
else
{
    for(int i=0; i<N; i+=16)
    {
        for(int j=0; j<M; j+=16)
        {
            for(int x=i; x<N&&x<i+16; x++)
            {
                for(int y=j; y<M&&y<j+16; y++)
                {
                    B[y][x]=A[x][y];
                }
            }
        }
    }
}
```

只是对其进行 16*16 分块，miss 数量就已经达到了要求。。。

原因是 A 矩阵是 61 列，不是什么 8 的倍数，以第二行为例，前几个数据会和第一行的末尾几个数据分到一个块里。这样天然错开使我们将矩阵扩大到 16*16 也不会出现之前那么大的冲突不命中压力。（如果对齐操作 16*16 将会几乎次次都要 miss 后驱逐），附上代码：

```
else
{
    for(int i=0; i<N; i+=16)
    {
        for(int j=0; j<M; j+=16)
        {
            for(int x=i; x<N&&x<i+16; x++)
            {
                for(int y=j; y<M&&y<j+16; y++)
                {
                    B[y][x]=A[x][y];
                }
            }
        }
    }
}
```

```
linleo@ubuntu:~/Desktop/cachelab-handout$ ./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:6187, misses:1992, evictions:1960

Summary for official submission (func 0): correctness=1 misses=1992

TEST_TRANS_RESULTS=1:1992
linleo@ubuntu:~/Desktop/cachelab-handout$
```

第 4 章 总结

4.1 请总结本次实验的收获

本次实验使我们对 cache 的内部结构有了十分深刻的认识，同时通过转置矩阵见识到了提高 cache 命中率的不易。

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.