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　计算机科学与技术

学　　号　1171000410

班　　级　1703005

学　　生　强文杰

指 导 教 师　吴锐

实 验 地 点　G712

实 验 日 期　2018.11.25

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 C S E B s e b

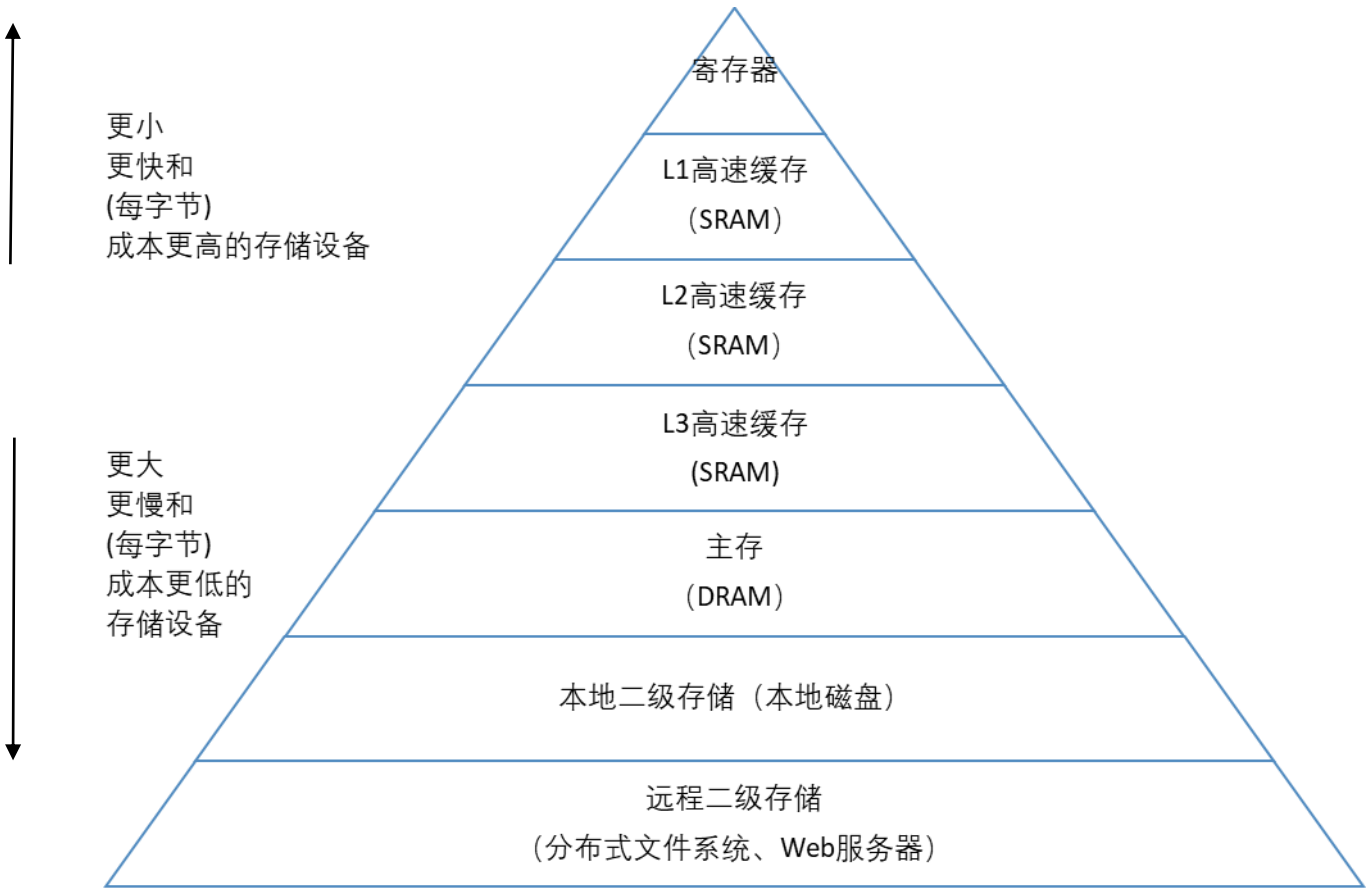
写出 Cache 的基本结构与参数

写出各类 Cache 的读策略与写策略

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）



从上到下分别为 L0 L1 L2 L3 L4 L5 L6 L6

高层的存储器保存着从底层的存储器取出的缓存行

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 cache：C=32KB　S=64　E=8　B=64B　s=6　e=3　b=6

二级 cache：C=256KB　S=1024　E=4　B=64B　s=10　e=2　b=6

三级 cache：C=3MB　S=4096　E=12　B=64B　s=12　e=4　b=6

2.3 写出各类 Cache 的读策略与写策略（5 分）

Cache 读策略

1：缓存命中，则从 cache 中读相应数据到 CPU 或上一级 cache 中。

2：缓存不命中，则从主存或下一级 cache 中读取数据，并替换出一行数据。

Cache 写策略

1、写回

当 CPU 写 Cache 命中时，只修改 Cache 的内容，而不是立即写入主存；只有当此块被换出时才写回主存。

2、直写

立即将一个已经缓存了的字 w 的高速缓存块写回到紧接着的第一层中。。

3、写分配

加载相应的低一层的块到高速缓存中，然后更新这个高速缓存块。

4、非写分配

避开高速缓存，直接把这个字写到低一层中去。

2.4 写出用 gprof 进行性能分析的方法（5 分）

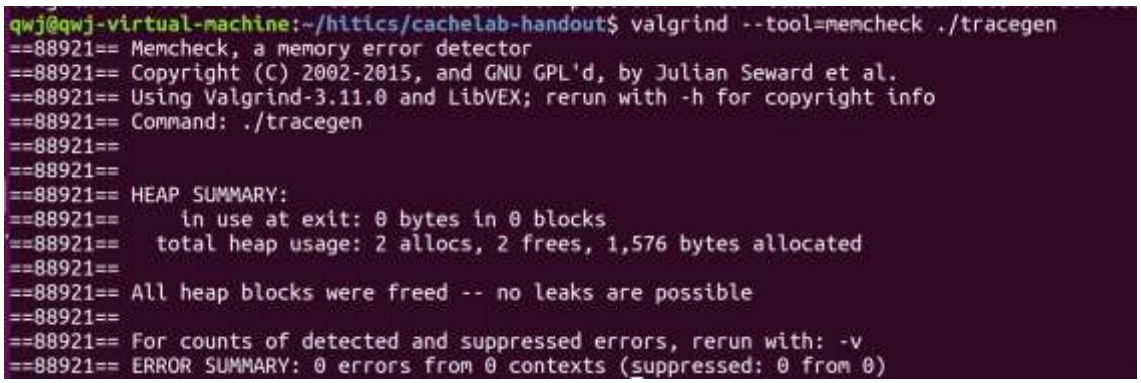
gprof 是 GNU profile 工具，可以运行于 linux、AIX、Sun 等操作系统进行 C、C++、Pascal、Fortran 程序的性能分析，用于程序的性能优化以及程序瓶颈问题的查找和解决。通过分析应用程序运行时产生的“flat profile”，可以得到每个函数的调用次数，每个函数消耗的处理器时间，也可以得到函数的“调用关系图”，包括函数调用的层次关系，每个函数调用花费了多少时间。使用步骤如下：

（1）用 gcc、g++、xlC 编译程序时，使用-pg 参数，如：g++ -pg -o test.exe test.cpp 编译器会自动在目标代码中插入用于性能测试的代码片断，这些代码在程序运行时采集并记录函数的调用关系和调用次数，并记录函数自身执行时间和被调用函数的执行时间。

（2）执行编译后的可执行程序，如：./test.exe。该步骤运行程序的时间会稍慢于正常编译的可执行程序的运行时间。程序运行结束后，会在程序所在路径下生成一个缺省文件名为 gmon.out 的文件，这个文件就是记录程序运行的性能、调用关系、调用次数等信息的数据文件。

（3）使用 gprof 命令来分析记录程序运行信息的 gmon.out 文件，如：gprof test.exe gmon.out 则可以在显示器上看到函数调用相关的统计、分析信息。上述信息也可以采用 gprof test.exe gmon.out> gprofresult.txt 重定向到文本文件以便于后续分析。

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。Valgrind 可以检测内存泄漏和内存违例，还可以分析 cache 的使用等。Valgrind 包含以下工具：Memcheck（用来检测程序中出现的内存问题，所有对内存的读写都会被检测到，一切对 malloc()/free()/new/delete 的调用都会被捕获）、Callgrind（收集程序运行时的一些数据，建立函数调用关系图，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（模拟 CPU 中的一级缓存 I1，Dl 和二级缓存，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）、Helgrind（用来检查多线程程序中出现的竞争问题）、Massif（堆栈分析器，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）。Valgrind 的使用非常简单，valgrind 命令的格式如下：valgrind [valgrind-options] your-prog [your-prog options] 。一些常用的选项如下：

选项	作用
-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	打印更详细的信息。
--tool= [default: memcheck]	最常用的选项。运行 valgrind 中名为 toolname 的工具。如果省略工具名，默认运行 memcheck。
--db-attach= [default: no]	绑定到调试器上，便于调试错误。

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：

1.首先介绍程序主要定义的变量和结构体：

```
typedef struct cache_line {
    char valid;          //有效位
    mem_addr_t tag;      //标识位
    int lru; //最后的访问时间距离现在最远的块
} cache_line_t;
```

cache_set_t; //储存每一组包含的行

cache_t; //定义指向组的指针

还有 s , E , hit_count , lru_counter 等全局变量就不再多说。

2.下面分析程序主要的函数：

在主函数中从命令行参数计算 S，E 和 B. 如下：

```
S = 1<<s;   //组数
B = 1<<b;   //块大小
E = E;
```

initCache()函数 - 分配内存，写 0 表示有效和标记和 LRU，为它们初始化

```
cache.sets = (cache_set_t*)malloc(S*sizeof(cache_set_t));
cache.sets = (cache_set_t*)malloc(S*sizeof(cache_set_t));//为组申请空间
```

cache.sets[i].lines = (cache_line_t*)malloc(E*sizeof(cache_line_t));//为行申请空间

freeCache()函数：为释放空间，根据申请空间的倒序来释放即可。

void replayTrace(char* trace_fn) ：此函数基本已经全部给出，主要的就是从 trace 文件中读取数据，并且调用 accessdata 函数，操作类型若为 'L'或 'S'，则调用一次 accessdata，若为 'M' ，则多调用一次 accessdata 。 另外在次函数中读取了地址 addr 之后，可以计算出组索引和标记：

set_index_mask =(addr>>b)&((1<<s)-1); //组索引

tag_mask = (addr>>b)>>s; //标记

3.单独介绍最重要的函数 accessdata

accessData - 访问内存地址 addr 的数据。

1)如果它已经在缓存中，则增加 hit_count

2)如果它不在缓存中，请将其放入缓存中，增加错过次数。

3)如果一条线被驱逐，也会增加 eviction_count

在函数中实现时，hit 发生的情况：组索引找到的某一组，存在一行有效位为 1，并且标记匹配。

若不 hit ，则直接 miss++。

再看是否驱逐，驱逐发生的情况为：组索引找到的某一组，有效位全部为 1，此时发生 evictions++ ，并且找到 lru 最小的那一行，驱逐。

另外，每次发生 hit 或者只 miss 或者 miss 加上 eviction ，都需要更新那一行的 lru 数值，具体的就是该行的 lru 取到最大，其他所有行的 lru 减一即可

```
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
# Generate a handin tar file each time you compile
tar -cvf qwj-handin.tar  csim.c trans.c
csim.c
trans.c
qwj@qwj-virtual-machine:~/hitics/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
```

测试用例 1 的输出截图（5 分）：

```
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9  misses:8  evictions:6
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ ./csim-ref -v -s 1 -E 1 -b 1 -t traces/yi2.trace
L 0,1 miss
L 1,1 hit
L 2,1 miss
L 3,1 hit
S 4,1 miss eviction
L 5,1 hit
S 6,1 miss eviction
L 7,1 hit
S 8,1 miss eviction
L 9,1 hit
S a,1 miss eviction
L b,1 hit
S c,1 miss eviction
L d,1 hit
S e,1 miss eviction
M f,1 hit hit
hits:9 misses:8 evictions:6
```

测试用例 2 的输出截图（5 分）：

```
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ ./csim -s 4 -E 2 -b 4 -t traces/yi2.trace
hits:16  misses:1  evictions:0
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ ./csim-ref -v -s 4 -E 2 -b 4 -t traces/yi2.trace
L 0,1 miss
L 1,1 hit
L 2,1 hit
L 3,1 hit
S 4,1 hit
L 5,1 hit
S 6,1 hit
L 7,1 hit
S 8,1 hit
L 9,1 hit
S a,1 hit
L b,1 hit
S c,1 hit
L d,1 hit
S e,1 hit
M f,1 hit hit
hits:16 misses:1 evictions:0
```

测试用例 3 的输出截图（5 分）：

```
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
hits:2  misses:3  evictions:1
qwj@qwj-virtual-machine:~/hitics/cachelab-handout$ ./csim-ref -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
```

测试用例 4 的输出截图（5 分）：

```
qwj@qwj-virtual-machine:~/hitics/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）：

```
qwj@qwj-virtual-machine:~/hitics/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）：

```
qwj@qwj-virtual-machine:~/hitics/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）：

```
qwj@qwj-virtual-machine:~/hitics/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）：

```
qwj@qwj-virtual-machine:~/hitics/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：

如果说实验的 cache 模拟是冷盘，那么矩阵才是大餐。

首先 。根据 ppt 的提示，使用模拟的 cache 对 trace[i]中对第 i 个函数的转置轨迹进行分析，这些访存轨迹文件对帮助调试和理解每一转置函数触发的缓存命中和缺失到底从何而来非常重要。至于为什么不适用 ./csim-ref ，是因为使用自己的模拟缓存，可以自定义函数的输出，打印组索引，标记位等关

键信息。

通过对组索引和标记的查看，可以得出：A 矩阵和 B 矩阵相同的下标映射到相同的组中，但是标记不同。这条结论是后面矩阵转置优化的前提。

第一步：32*32 。因为 A 矩阵和 B 矩阵中下标相同的元素会映射到一个块中，因此访问两个数组的过程中会发生很多的冲突不命中。因此要减少 miss，必须减少冲突不命中。基于这个原因，可以想到，我们可以一次性访问一个块中的多个元素，访问完就不再访问这个块了。

因为 cache 一个 block 中可以存 8 个 int 数据，因此先考虑 8*8 将其分块，这样分块的一个更重要的原因是当我将 8 个 int 直接取出来，这样即使写入的时候替换了 block 也没关系，因为我们已经全部读入了。

```
if(M==32&&N==32)
{
    for(i=0;i<M;i=i+8)
    {
        for(j=0;j<N;j++)
        {
            temp1 = A[j][i];
            temp2 = A[j][i+1];
            temp3 = A[j][i+2];
            temp4 = A[j][i+3];
            temp5 = A[j][i+4];
            temp6 = A[j][i+5];
            temp7 = A[j][i+6];
            temp8 = A[j][i+7];
            B[i][j] = temp1;
            B[i+1][j] = temp2;
            B[i+2][j] = temp3;
            B[i+3][j] = temp4;
            B[i+4][j] = temp5;
            B[i+5][j] = temp6;
            B[i+6][j] = temp7;
            B[i+7][j] = temp8;
        }
    }
}
```

事实上，对于测试出来的 287 次 miss，我们也能有如下的分析：对于 A 数组而言，每块只有第一个元素 miss，这也是无法避免的；对于 B 数组而言，除了每块第一个元素不命中，矩阵对角线上的元素也会不命中。因此 miss 位 287 次依然有优化空间，在此没有展开。

第二步 64*64 。对于 64*64 的矩阵而言，每一行元素会占 8 个组，因此 4 行元素即可占满 cache。

我先尝试着和 32*32 的矩阵相同的分块方法，发现 miss 总数达到了 4000+，基本没什么优化。仔细分析应该有如下的原因：对 A 数组的访问依然是第一个不命中。对 B 数组的访问，可以看到前 4 行和后四行所映射的块是相同的，于是访问完前四行的第一列后，访问后四行的第一列会冲突不命中，导致原来的块被驱逐，再访问前四行的第二列，由于之前的块已经被驱逐，因此又会 miss 且驱逐，如此反复下去，B 数组中所有的元素皆会不命中。

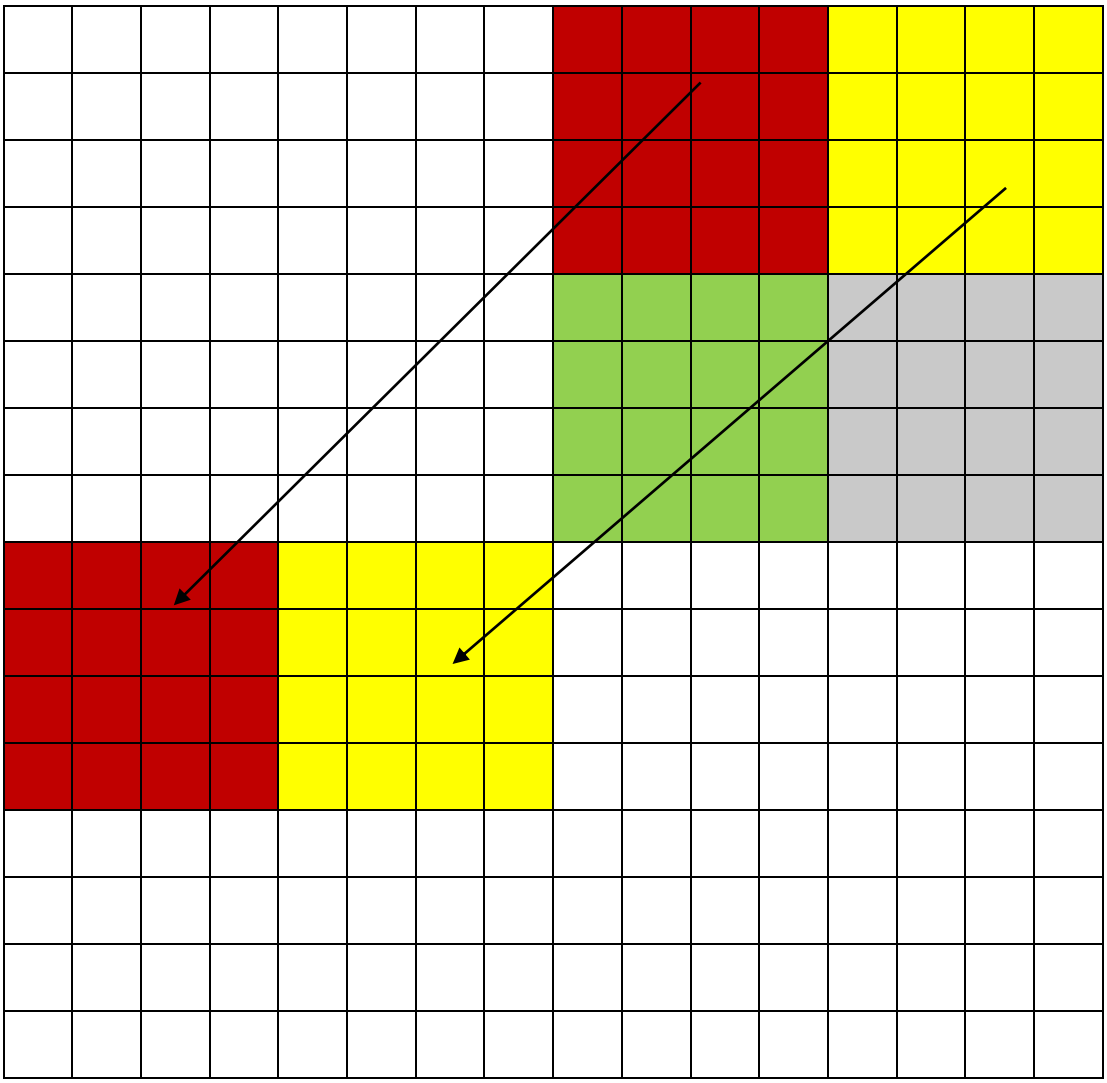
由以上否定了 8*8 的划分方式。

又因为 4*4 的分块方式无法充分利用每次加载后的块，故也将其否定。

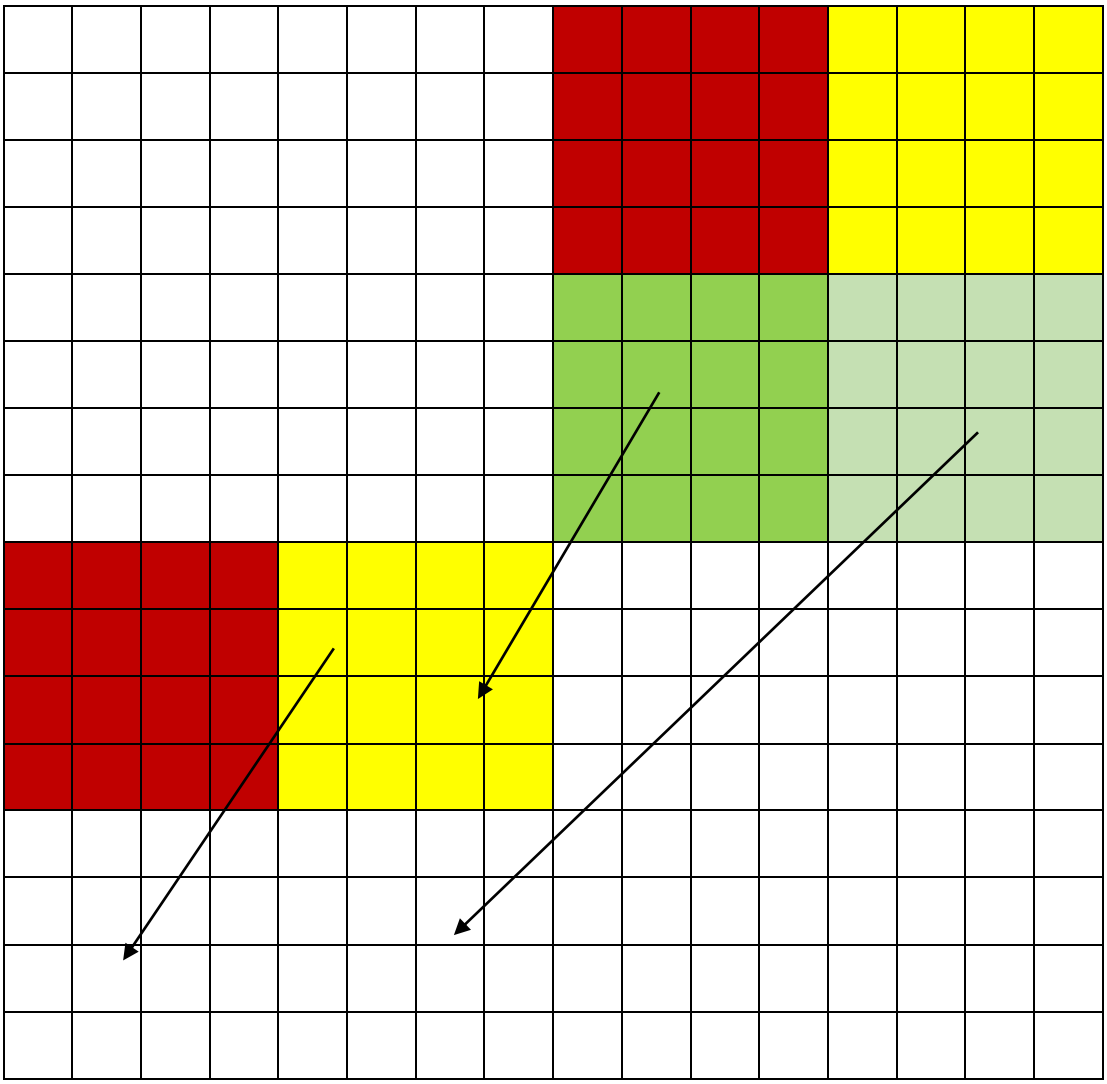
由此可以想到一个折中的办法，就是将 8*8 和 4*4 相结合。

下面用示意图进行表示：

首先将红色块移至目的地，再将黄色块移至红色块左边“暂存”一下。此时移动是伴随着转置的。



再实现图中的移动即可，将黄色块移到红色块的下面，在将绿色快移到之前“暂存”黄色块的地方，最后将灰色块移动到目的地即可。



根据以上的示意图可以写出如下的过程：

```
if(M==64&&N==64)
{
    for(i=0;i<N;i=i+8)
    {
        for(j=0;j<M;j=j+8)
        {
            for(k=i;k<i+4;k++)
            {
                temp1=A[k][j];
                temp2=A[k][j+1];
                temp3=A[k][j+2];
                temp4=A[k][j+3];
                temp5=A[k][j+4];
                temp6=A[k][j+5];
                temp7=A[k][j+6];
                temp8=A[k][j+7];
                B[j][k]=temp1;
                B[j+1][k]=temp2;
                B[j+2][k]=temp3;
                B[j+3][k]=temp4;
                B[j][k+4]=temp5;
                B[j+1][k+4]=temp6;
                B[j+2][k+4]=temp7;
                B[j+3][k+4]=temp8;
            }
            for(p=j;p<j+4;p++)
            {
                temp1=A[i+4][p];
                temp2=A[i+5][p];
                temp3=A[i+6][p];
                temp4=A[i+7][p];
                temp5=B[p][i+4];
                temp6=B[p][i+5];
                temp7=B[p][i+6];
                temp8=B[p][i+7];
                B[p][i+4]=temp1;
                B[p][i+5]=temp2;
                B[p][i+6]=temp3;
                B[p][i+7]=temp4;
                B[p+4][i]=temp5;
                B[p+4][i+1]=temp6;
                B[p+4][i+2]=temp7;
                B[p+4][i+3]=temp8;
            }
            for(k=i+4;k<i+8;k++)
            {
                temp1=A[k][j+4];
                temp2=A[k][j+5];
                temp3=A[k][j+6];
                temp4=A[k][j+7];
                B[j+4][k]=temp1;
                B[j+5][k]=temp2;
                B[j+6][k]=temp3;
                B[j+7][k]=temp4;
            }
        }
    }
}
```

最后 61*67 。对于之前的两个矩阵，由于它们的每一行元素恰好占整数个块，因此分块的时候也会利用这一特性。但是根据之前的处理，由于这个测试的 miss 数允许到 2000，这个限度相对较大，因此我们可以尝试不同的分块来处理即可。因此相当于 64*64 需要反复优化，这个还是比较好处理的。

根据反复调整分块的大小，发现分成 17*17 的块时 miss 数最小，代码如下：

```
    if(M==61&&N==67)
    {
        for(i=0;i<N;i=i+17)
        {
            for(j=0;j<M;j=j+17)
            {
                for(k=i;k<i+17 && k<N;k++)
                {
                    for(p=j;p<j+17 && p<M;p++)
                    {
                        temp1 = A[k][p];
                        B[p][k] = temp1;
                    }
                }
            }
        }
    }
}
```

```
qwj@qwj-virtual-machine:~/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        1950
          Total points    53.0        53
```

32×32（10 分）：运行结果截图

```
qwj@qwj-virtual-machine:~/hitics/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
^[[A# Generate a handin tar file each time you compile
tar -cvf qwj-handin.tar  csim.c trans.c
csim.c
trans.c
qwj@qwj-virtual-machine:~/hitics/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
```

64×64（10 分）：运行结果截图

```
qwj@qwj-virtual-machine:~/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
```

61×67（20 分）：运行结果截图

```
qwj@qwj-virtual-machine:~/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:6229, misses:1950, evictions:1918
```

第 4 章 总结

4.1 请总结本次实验的收获

理解现代计算机存储器层次结构
学会自己完成对 cache 的模拟，深入理解 cache 的读写策略和替换策略等
学会利用分块策略分析问题
掌握 linux 下性能测试方法和技巧

4.2 请给出对本次实验内容的建议

对于 64*64 的矩阵转置，难度稍大，个人认为 ppt 上可以多给些提示

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.