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 计算学部

学　　号 _____

班　　级 _____

学　　生 _____

指 导 教 师 _____

实 验 地 点 G709

实 验 日 期 2021.5.27

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

1.2.2 软件环境

Windows10 64 位; Vmware 11；Ubuntu 16.04 LTS 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

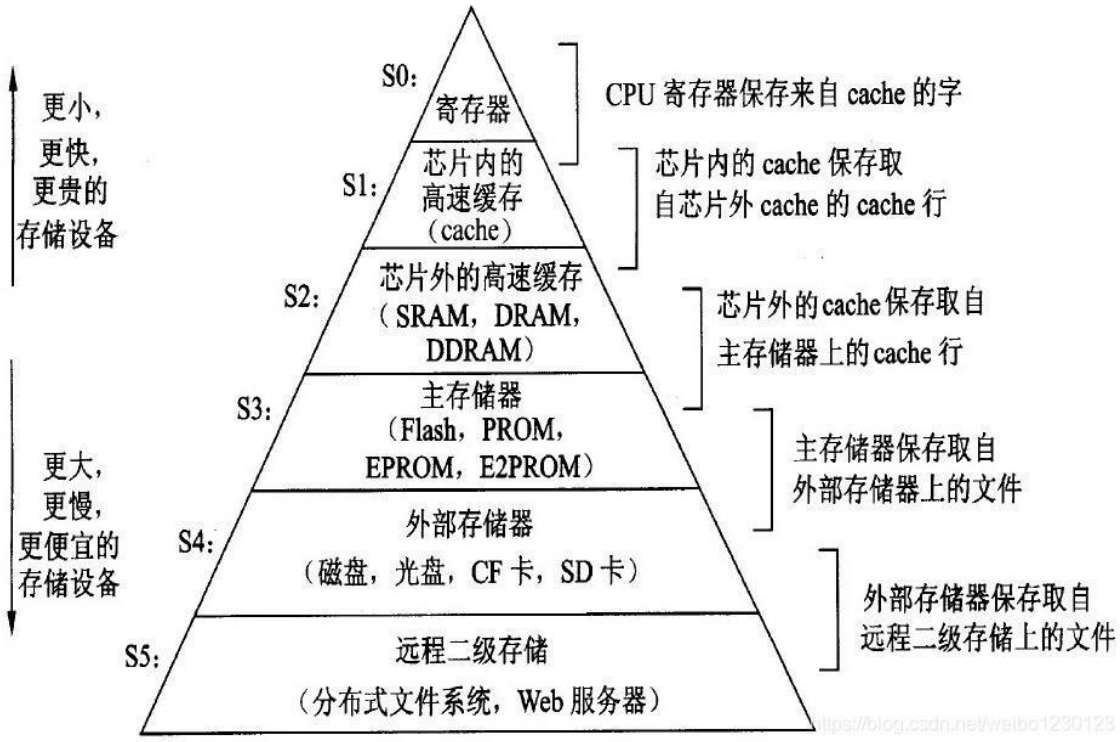
上实验课前，必须认真预习实验指导书（PPT 或 PDF）

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）



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）

一级数据缓存 C: 192KB S:384 E:8 B:64 s:9 b:6 e:3

一级指令缓存 C: 192KB S:384 E:8 B:64 s:9 b: 6 e:3

二级缓存 C:1536KB S:6144 E:4 B:64 s:13 b: 6 e:2

三级缓存 C:12MB S:12288 E:16 B:64 s:14 b:6 e:4

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，从对应的 cache 向 CPU 中的寄存器文件传送数据或是向上一级 cache 传送数据。
2. 缓存未命中，从主存或是下一级 cache 读取需要的数据，并根据情况决定是否要驱逐数据来写入需要的数据。

Cache 写策略：

1. 写命中：
   a. 直写：立即将 w 的高速缓存快写回到紧接着的第一层中
   b. 写回：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第一层中
2. 写不命中
   a. 写分配，加载相应的第一层的块到高速缓存中，然后更新这个高速缓存块
   b. 非写分配，避开高速缓存，直接把这个字写到低一层中。

2.4 写出用 gprof 进行性能分析的方法（5 分）

gprof 是 GNU 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 包含下列工具：

1、memcheck：检查程序中的内存问题，如泄漏、越界、非法指针等。

2、callgrind：检测程序代码的运行时间和调用过程，以及分析程序性能。

3、cachegrind：分析 CPU 的 cache 命中率、丢失率，用于进行代码优化。

4、helgrind：用于检查多线程程序的竞态条件。

5、massif：堆栈分析器，指示程序中使用了多少堆内存等信息。

6、lackey：

7、nulgrind：

这几个工具的使用是通过命令：valgrand --tool=name 程序名来分别调用的，当不指定 tool 参数时默认是 --tool=memcheck

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：

由于程序的主体框架已经给定了，因此我们需要实现的部分主要就是 cache 的初始化、释放空间以及模拟 cache 的查询数据的过程。

首先是初始化函数，具体的函数代码如下：

```
//todo...
  int i, j;
  if (s < 0)
  {
      printf("error");
      exit(0);
  }
  cache = (cache_set_t*)malloc(sizeof(cache_set_t) * S);
  if (cache == NULL)
  {
      printf("no set space");
  }
  for (int i = 0; i < S; i++)
  {
      cache[i] = (cache_set_t)malloc(sizeof(cache_line_t) * E);
      if(cache[i]==NULL)printf("no set space");
  }
  for (int i = 0; i < S; i++)
  {
      for (int j = 0; j < E; j++)
      {
          cache[i][j].valid = 0;
          cache[i][j].tag = 0;
          cache[i][j].lru = 0;
      }
  }
  set_index_mask = (1 << s) - 1;
```

可以发现我们是将 cache 视为一个二维的指针数组，其中第一维是 S 个

cache_set_t 类型的指针代表 S 个组，而第二维有 E 个 cache_line_t 代表每个组有 E 行，这样就基本形成了一个 cache，接着我们对于分配空间中的每一个空间都设置了 valid，tag，lru，其中我们将掩码 set_index_mask 设置为（1<<s）-1，主要的目的是为了后面获得 tag 的时候用一个移位和按位与就可以得到我们需要的 cache 中行的 tag。

接着是 free 函数，这个函数相对比较简单，主要就是使用了 c 语言的函数 free 将 init 中申请的空间释放。

最后是最关键的一个函数——accessData，这个函数也是模拟 cache 行为的最关键的函数，主要的思路就是首先求出我们需要取的数据的 tag 和 set，如果在 cache 中存在这个数据的话，那么就意味着命中，那么就可以将对应的数+1，同时由于我们使用 lru 来标志每一个数据有多久没有被使用过了，因此命中一次之后就将该数据的 lru 设置为 0；如果未命中的话，需要考虑是否需要驱逐，首先我们先找到 lru 最大的数，这个数据就是有可能需要被驱逐的数据。分为两种情况，一种是冷不命中，这种情况对应需要写入的数据位置的 valid 等于 0，而需要驱逐的情况就对应着 valid 等于 1。不命中情况的代码如下：

```
if (!flag)
{
    int evictionData=0;
    miss_count++;
    if (verbosity)printf("miss");
    int max = 0;
    for (int i = 0; i < E; i++)
    {
        if (cache[set][i].lru > max)
        {
            max = cache[set][i].lru;
            evictionData = i;
        }
    }
    cache[set][evictionData].tag = tag;//设置标记位
    if (cache[set][evictionData].valid == 1)//需要驱逐
    {
        eviction_count++;
        if (verbosity)printf("eviction");
    }
    else//不需要驱逐
    {
        cache[set][evictionData].valid = 1;
    }
    for (int i = 0; i < E; i++)
    {
        cache[set][i].lru++;
    }
    cache[set][evictionData].lru = 0;
}
```

测试用例 1 的输出截图（5 分）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -s 1 -E 1 -b
1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
zsh@zsh-virtual-machine:~/code/c/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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 4 -E 2 -b 4 -t
 traces/yi.trace
hits:4 misses:5 evictions:2
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 2 -E 1 -b 4 -t
 traces/dave.trace
hits:2 misses:3 evictions:1
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 2 -E 1 -b 3 -t
 traces/trans.trace
hits:167 misses:71 evictions:67
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 2 -E 2 -b 3 -t
 traces/trans.trace
hits:201 misses:37 evictions:29
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 2 -E 4 -b 3 -t
 traces/trans.trace
hits:212 misses:26 evictions:10
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t
 traces/trans.trace
hits:231 misses:7 evictions:0
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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）：

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t
 traces/long.trace
hits:265189 misses:21775 evictions:21743
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./csim-ref -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：

首先我们需要考虑为什么需要对矩阵转置的程序进行不同方式的优化，这是因为我们知道 cache 的参数为 32 组，每组 1 行，每行可存放 32 个字节。也就是说 cache 中一行可以存储 8 个 int 型的数据，如果我们采用最简单的二维数组直接交换的话 A 和 B 在 cache 中总是共享同一个块，那么很显然出现的 miss 的情况将会非常多，这就是我们需要进行优化的原因。

1.32*32

我们观察到 cache 中每块可以存储 8 个 int 型的数据，那么也就是矩阵的 8 行可以填满一个 cache，也就是说矩阵中相差 8 行的数据是共享一个 cache 的，基于这个事实我们将矩阵分割成 8*8 的小矩阵，可以看如下示意图：

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

其中我们把每一个格子视为一个 8*8 的矩阵，以编号为 2 的矩阵为例，我们

需要做的是将它放到编号为 5 的矩阵的位置，并进行转置，很显然，这两个小矩阵是不会共享一个块的，那么出现的不命中的情况就会大大下降。其余的格子类似。这种方法中主要出现的 miss 情况出现在对角线格子上，在这些小矩阵转置的时候 A 和 B 是共享一个块的，因此这是可以优化的，但是由于不优化的情况已经达到满分条件，因此不再优化，主要代码截图如下：

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

2.64*64

在 64*64 的情况下我们可以再次尝试一下 8*8 的分割法，发现 miss 的很多。这是因为在这种情况下每 4 行就会填满一次 cache，那就意味着隔四行的两个数据是共享一个块的，再使用 8*8 势必导致前四行与后四行共享一个块，就会导致 miss 的情况大大增加。那么是不是可以使用 4*4 呢？显然也是不可以的，由于 cache 中每一行可以填写 8 个 int 型的数据，如果 4*4 意味着矩阵中相邻的两个小矩阵可能会共享一个块，这样就导致了效果不是很好。因此我们想到了使用 4*8 的情况，由于我们选择的块需要是一个方形，因此我们采用 4*4 和 8*8 相结合的方法进行读取。

不妨使用如下表格来表示矩阵（不完整，只是一部分）：

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

假设每一个小格子都是 4*4 的小矩阵，假设我们现在需要把 3 号矩阵转置放到 9 号处，那根据上述的分析，我们一次性处理 3 号和 4 号矩阵，3 号直接放到 9 号并转置，4 号可以暂时放在 10 号矩阵处，并进行转置。这种方法就解决了 4*4 情况下对于 cache 空间使用不充分带来的 miss 数上升的问题。接下来由于 4 号最后是需要放到 13 号的，10 号应该放的是 7 号矩阵的转置。那么我们接下来就处理这个，我们将 10 号矩阵当前的内容（也就是 4 号的转置结果）放置到 13 号矩阵中，7 号矩阵的内容转置放到 10 号，8 号矩阵的内容转置放到 14 号。这个过程就很好地解决了 4*4 和 8*8 各自的不足，经过测试，miss 数已经小于 1300。主要代码截图如下：

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

```
            int temp5=B[m][i+4];
            int temp6=B[m][i+5];
            int temp7=B[m][i+6];
            int temp8=B[m][i+7];
            B[m][i+4]=temp1;
            B[m][i+5]=temp2;
            B[m][i+6]=temp3;
            B[m][i+7]=temp4;
            B[m+4][i]=temp5;
            B[m+4][i+1]=temp6;
            B[m+4][i+2]=temp7;
            B[m+4][i+3]=temp8;
        }
        for(int k=i+4;k<i+8;k++)
        {
            int temp1=A[k][j+4];
            int temp2=A[k][j+5];
            int temp3=A[k][j+6];
            int 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;
        }
    }
}
```

3.61*67

因为这一情况与上述不同，不是一个规则的矩阵，也不是很容易分为方阵的组合，因此这一问的解决方法主要是使用尝试的方式，发现分为 17*17 的方阵的时候 miss 的数量满足小于 3000 的条件，因此使用这一结果。

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

32×32（10 分）：运行结果截图

```
zsh@zsh-virtual-machine:~/code/c/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:1765, misses:288, evictions:256

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:869, misses:1184, evictions:1152

Summary for official submission (func 0): correctness=1 misses=288

TEST_TRANS_RESULTS=1:288
```

64×64（10 分）：运行结果截图

```
zsh@zsh-virtual-machine:~/code/c/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:9065, misses:1180, evictions:1148

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:3473, misses:4724, evictions:4692

Summary for official submission (func 0): correctness=1 misses=1180

TEST_TRANS_RESULTS=1:1180
```

61×67（20 分）：运行结果截图

```
zsh@zsh-virtual-machine:~/code/c/lab6/cachelab-handout$ ./test-trans -M 61 -N 6
7

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:6228, misses:1951, evictions:1919

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:3755, misses:4424, evictions:4392

Summary for official submission (func 0): correctness=1 misses=1951

TEST_TRANS_RESULTS=1:1951
```

第 4 章 总结

4.1 请总结本次实验的收获

这次实验之后对于 cache 的基本工作机制了解更加深入了，同时对于如何编写一个高速缓存友好的代码有了更深入的了解。

4.2 请给出对本次实验内容的建议

可以在 PPT 上增加一些对于常见错误的解释，比如不能在共享文件夹下操作。

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.