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 1183200123

班　　级 1803003

学　　生 祁 天

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2019.11.27

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 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

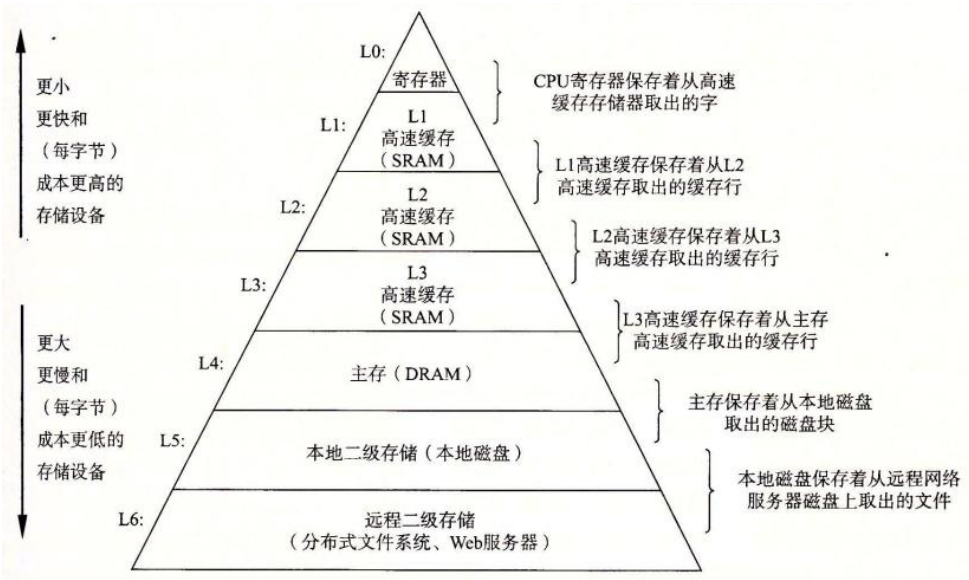
1.3 实验预习

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。

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）

一级缓存大小：32KB，8 路组相联，每块 64 字节

C(字节)	S	E	B	s	e	b
32768	64	8	64	6	3	6

二级缓存大小：256KB，4 路组相联，每块 64 字节

C(字节)	S	E	B	s	e	b
262144	1024	4	64	10	2	6

三级缓存大小：8MB，16 路组相联，每块 64 字节

C(字节)	S	E	B	s	e	b
8388608	8192	16	64	13	4	6

2.3 写出各类 Cache 的读策略与写策略（5 分）

2.3.1 读策略

缓存命中：直接从该层读取数据

缓存不命中：替换策略，其中“随机替换策略”会随机选择一个牺牲块；“最近最少被使用（LRU）替换策略”会选择那个最后被访问的时间距现在最远的块。

2.3.2 写策略

写命中：

a. 直写，就是立即将已经缓存了的字的高速缓存块写回到紧接着的低一层中；

b. 写回：尽可能地推迟更新，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。

写不命中：

a. 写分配，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；

b. 非写分配，避开高速缓存，直接把这个字写到低一层中。

直写高速缓存通常是非写分配的，写回高速缓存通常是写分配的。

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
```

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));//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
```

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

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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）：

```
qt1183200123@ubuntu:~/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：

首先，分析给定的缓存结构：s=5，E=1，b=5, S = 2 ^ 5 = 32 , B = 2 ^ 5 = 32

C = E * S * B = 1024 Byte ，对于 int 数组，可以存下 1024 / 4 = 256 个数组元素。每一组仅有一行（块），每块 32 个字节，即 32 / 4 = 8 个数组元素。要使 miss 数尽量小，即使几次连续的访存的对象尽量是内存中连续的数组元素。并且，需要特别注意的是：考虑到缓存最多存下 256 个数组元素，所以，地址序号 MOD 256 相等的内存块，映射到缓存的同一块。下面就不同大小的矩阵进行分别分析：

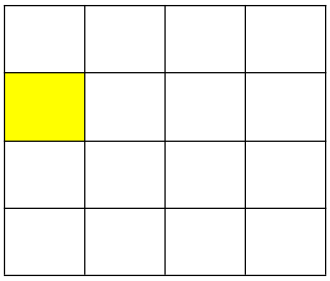
1. 32×32

如果按照例程中的访问方法：原矩阵整行整行的访问，伴随着目标矩阵整列整列的访问。根据上述对缓存的分析，整个缓存大小最多存下 256 个数组元素，即本矩阵中的 256 / 32 = 8 行。也就是说，整个缓存最多存 8 行。所以，整列整列访问的数组，每访问完 8 行，接下来的 8 行访问总会驱逐掉上 8 行的缓存块，等到访问下一个整列的时候，以前访问过的行，又会发生缓存不命中，于是，整个数组全部都不命中，（这个数组的）不命中率为 100%

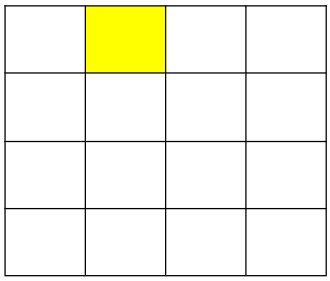
据此分析，我们想要：整列整列访问的数组，不再整列整列访问，而是充分利用其缓存的每 8 行。于是，就想到使用 8×8 分块的方法：把矩阵分为 16 个 8×8 的块。遍历每一块。在每一块中，遍历块中的每一个元素。这样，需要 4 个循环变量。我们还有 8 个局部变量可用，可以用来存储每一行的 8 个元素，以便一次性访问该缓存块中的元素，把不命中率降到最低。这样，不命中数就只剩下每个缓存块开始的冷不命中，即 1 / 4 的不命中率。考虑到对角线上的情况，还会存在 32 个额外的不命中率。所以，估计大概的不命中数为 ： 32 * 32 / 4 + 32 = 288 符合要求。

2. 64×64

与 32×32 块不同，缓存 256 个元素的容量，只能存下这个数组的 4 行。但如果照搬上面的思想，相应地采用 4×4 的分块方式，分成 256 块。考虑最理想的情况：每个小块只有 4 * 2 = 8 次不命中（乘 2 是因为读+写），一共有 8 * 256 = 2048 次不命中，这还是不考虑对角线的情况，显然不符合要求。所以，又想到 8×8 分块的方法。可是，怎么解决在每小块中，整列访问时，后 4 行对前 4 行缓存的驱逐呢？想到以下方法：



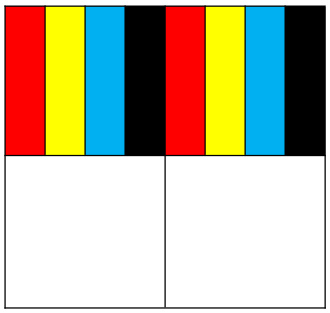
原矩阵



目的矩阵

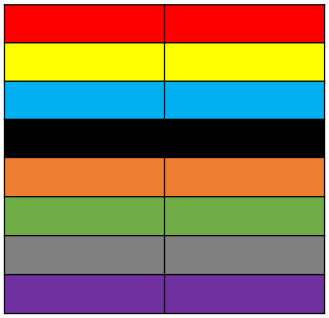
以图中的黄色 8×8 块为例，放大如下：

原矩阵的一个 8×8 块



目的矩阵的一个 8×8 块

如上图所示，同一颜色的表示，原矩阵中的一行，或目的矩阵原本在一起的一列。把原本放在“下 4 行”的列“暂存”在“上 4 行”。这样，就充分利用了前 4 行的缓存。接下来，需要把“错位”的目的矩阵“归位”：



原矩阵的一个 8×8 块

目的矩阵的一个 8×8 块

归位过程中，同样不会造成缓存的驱逐。最后，只需把右下角一小块搞定就行。值得注意的是：同 32 * 32 矩阵，在复制的全过程中，都需要局部变量起到暂存的作用，以便一次性访问该缓存块中的元素。

3. 61×67

这里由于矩阵的规模为 61×67，所以对于各个元素所在块的规律会较之前有所不同。对于之前的 32×32 以及 64×64，由于每一行的元素个数恰好为 8 的倍数，所以每一行恰好会占满整数个块，那么在编写代码的过程中就很容易利用这一特性。

而对于 61×67 规模而言，这里每一行不再满足这样的规律。例如第一行的最后面 5 个元素和第二最前面 3 个元素是在同一个块的。由于这个特性，对于整个矩阵而言，各个元素所在块的情况就变得不好处理，分析起来也比较复杂。于是，我尝试多种分块方法，发现 16*16 的分块法可以达到效果。分块余出的元素，单独处理即可。

32×32（10 分）：运行结果截图

```
qt1183200123@ubuntu:~/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
```

64×64（10 分）：运行结果截图

```
qt1183200123@ubuntu:~/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

```
qt1183200123@ubuntu:~/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

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=1950

TEST_TRANS_RESULTS=1:1950
```

总结图：

```
qt1183200123@ubuntu:~/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
```

第 4 章 总结

4.1 请总结本次实验的收获

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.