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 1170500913

班　　级 1703002

学　　生 熊健羽

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2018.11.27

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

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64 位)
GPU：Intel(R) HD Graphics 620
Nvidia GeForce 940MX
物理内存：16.00GB
磁盘：1TB HDD
128GB SSD

1.2.2 软件环境

Windows10 64 位；
Vmware 14.11；
Ubuntu 18.04 64 位；

1.2.3 开发工具

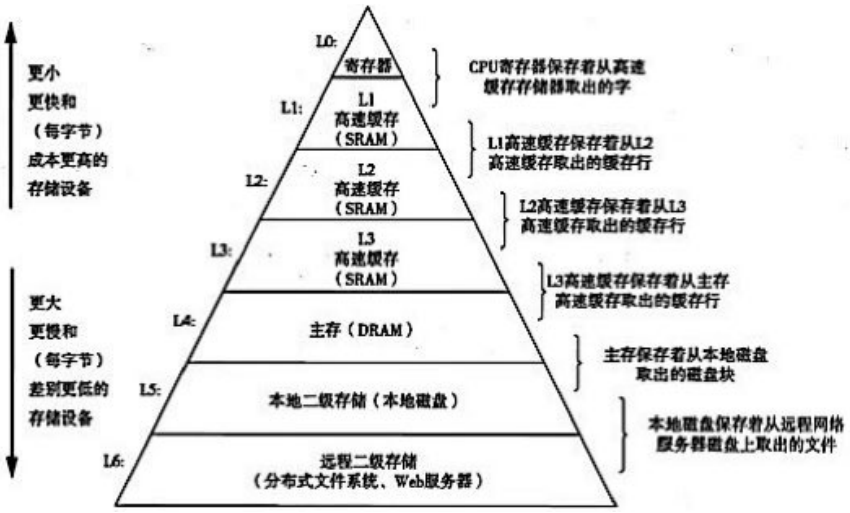
Visual Studio 2010 64 位；
Code::Blocks；
gedit，gcc，notepad++；

1.3 实验预习

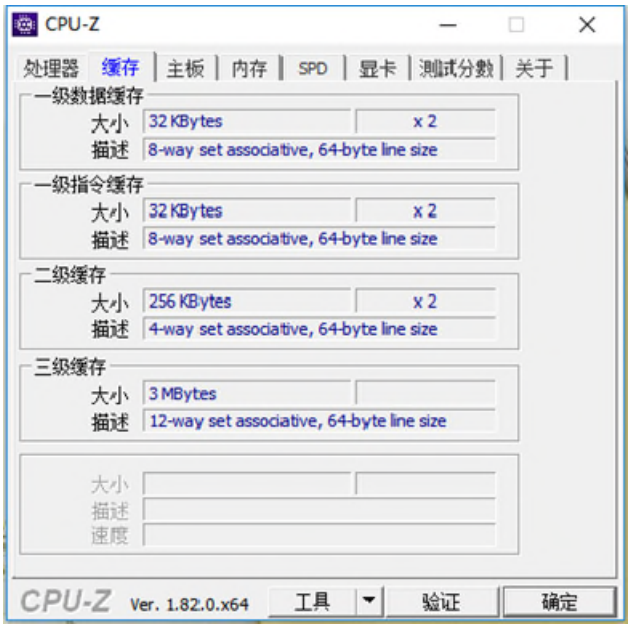
填写

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：32KB 8 路组相联，64 字节每块。

C(字节)	S	E	B	s	b
32768	64	8	64	6	6

二级缓存：256KB 4 路组相联，64 字节每块。

C(字节)	S	E	B	s	b
262144	1024	4	64	10	6

三级缓存：6MB 12 路组相联，64 字节每块。

C(字节)	S	E	B	s	b
6291456	4096	12	64	12	6

2.3 写出各类 Cache 的读策略与写策略（5 分）

1）缓存命中：直接从该层读取数据
2）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。
写策略：1）写命中 2）写不命中
1）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。
2）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）

在编译时加上-pg 选项，编译器就会在编译程序时在每个函数的开头加一个 mcount 函数调用，在每一个函数调用之前都会先调用这个 mcount 函数，在 mcount 中会保存函数的调用关系图和函数的调用时间和被调次数等信息。最终在程序退出时保存在 gmon.out 文件中，需要注意的是程序必须是正常退出或者通过 exit 调用退出，因为只要在 exit（）被调用时才会触发程序写 gmon.out 文件。

那么，gprof 的使用方法主要以下三步：

- 用-pg 参数编译程序
- 运行程序，并正常退出
- 查看 gmon.out 文件

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，如 Memcheck,Cachegrind,Helgrind,Callgrind，Massif。
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。
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

LOG 信息输出：
-xml=yes 将信息以 xml 格式输出，只有 memcheck 可用
-num-callers=<number> show <number> callers in stack traces [12]
-error-limit=no|yes 如果太多错误，则停止显示新错误? [yes]
-error-exitcode=<number> 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=<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|summary|full 要求对 leak 给出详细信息? [summary]
-leak-resolution=low|med|high how much bt merging in leak check [low]
-show-reachable=no|yes show reachable blocks in leak check? [no]

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. main 函数（调用 initCache 的前面一小部分）：

   根据输入的参数：s，b。计算出组数 S，每行的块数 B

2. initCache 函数：

   首先，给整个缓存分配出 S 组的空间；其次，分别给 S 个组分配出 E 行的空间：

```
void initCache()
{
    cache = (cache_t)calloc(S, sizeof(cache_set_t));
    for (int i = 0; i < S; i++)
        cache[i] = (cache_set_t)calloc(E, sizeof(cache_line_t));
}
```

3. freeCache 函数：

   首先，释放每个组中的各行的空间；其次，分别释放每个组的空间。

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

4. accessData 函数：

   首先，根据访问地址，确定组号、标记号，确定到某一组。

   首先查找，是否存在某一行，tag 等于待访问的标记号，且 valid 为 1：

a) 缓存命中：访问成功，hit_count 加 1，相应块的 lru 值更新为当前值。

b) 缓存不命中：miss_count 加 1，查找是否有空块（即搜索是否有块的 valid 值为 0）。若有，则该空块的 valid 置为 1、tag，置为标记号，相应块的 lru 值更新为当前值。若无空块，驱逐数 eviction_count 加 1，然后搜索组中的所有的行，找到 lru 值最小的（即上次访问距离最久的块），该块的 valid 置为 1、tag，置为标记号，相应块的 lru 值更新为当前值。

最后，访问计数器 lru_counter 加 1。

测试用例 1 的输出截图（5 分）：

```
xjy1170500913@ubuntu:~/桌面/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）：

```
xjy1170500913@ubuntu:~/桌面/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）：

```
xjy1170500913@ubuntu:~/桌面/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）：

```
xjy1170500913@ubuntu:~/桌面/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）：

```
xjy1170500913@ubuntu:~/桌面/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）：

```
xjy1170500913@ubuntu:~/桌面/cachelab-handout$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
hits:212 misses:26 evictions:10
```

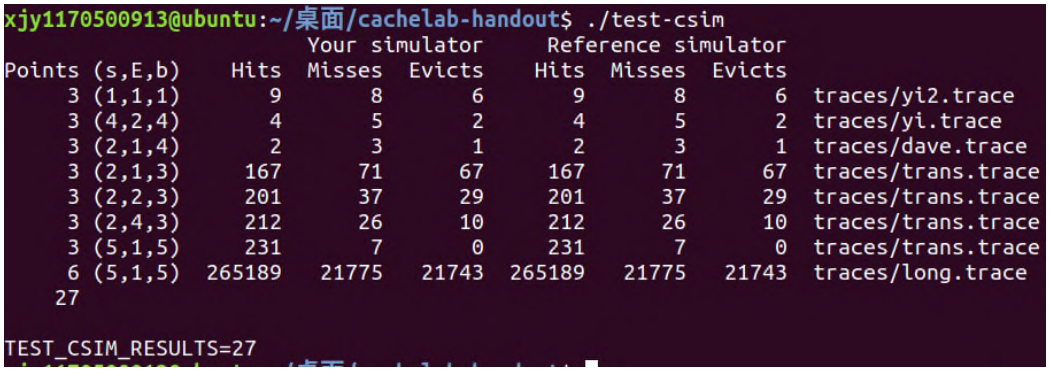
测试用例 7 的输出截图（5 分）：

```
xjy1170500913@ubuntu:~/桌面/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）：

```
xjy1170500913@ubuntu:~/桌面/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
```

总截图：



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

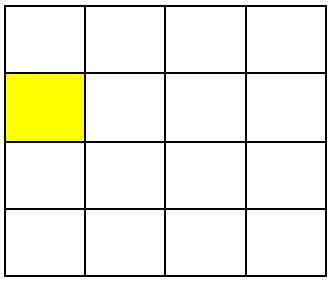
   如果按照例程中的访问方法：原矩阵整行整行的访问，伴随着目标矩阵整列整列的访问。根据上述对缓存的分析，整个缓存大小最多存下 256 个数

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%

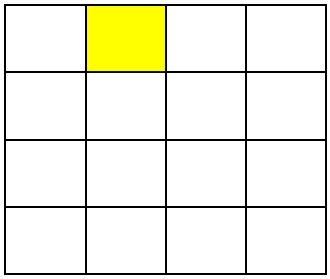
据此分析，我们想要：整列整列访问的数组，不再整列整列访问，而是充分利用其缓存的每 8 行。于是，就想到使用 8×8 分块的方法：把矩阵分为 16 个 8×8 的块。遍历每一块。在每一块中，遍历块中的每一个元素。这样，需要 4 个循环变量。我们还有 8 个局部变量可用，可以用来存储每一行的 8 个元素，以便一次性访问该缓存块中的元素，把不命中率降到最低。这样，不命中数就只剩下每个缓存块开始的冷不命中，即 1 / 4 的不命中率。考虑到对角线上的情况，还会存在 32 个额外的不命中率。所以，估计大概的不命中数为 ： 32 * 32 / 4 + 32 = 288 符合要求。

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：

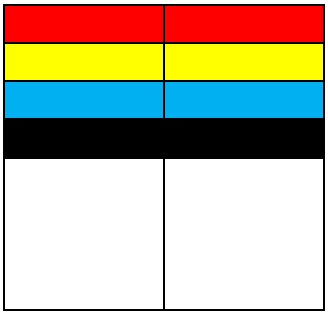


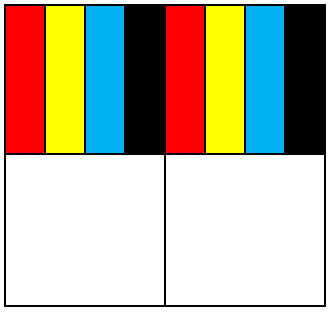
原矩阵



目的矩阵

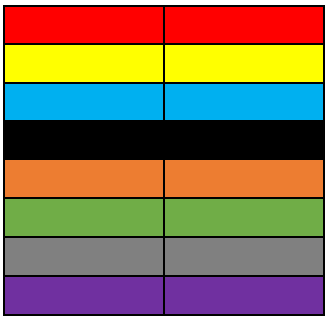
以图中的黄色 8×8 块为例，放大如下：



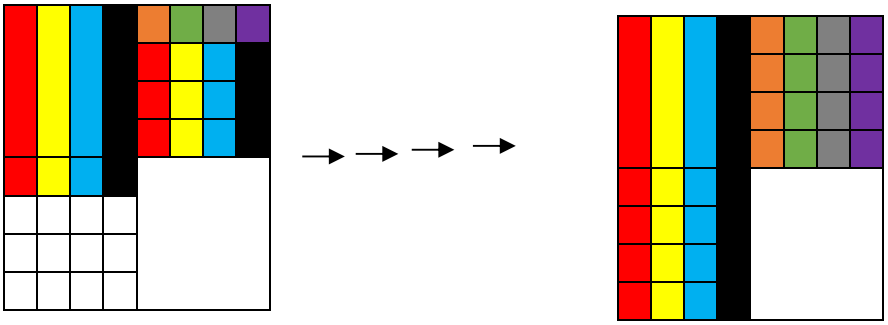


原矩阵的一个 8×8 块　　目的矩阵的一个 8×8 块

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

```
xjy1170500913@ubuntu:~/桌面/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
```

64×64（10 分）：运行结果截图

```
xjy1170500913@ubuntu:~/桌面/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:9018, misses:1227, evictions:1195
```

61×67（20 分）：运行结果截图

```
xjy1170500913@ubuntu:~/桌面/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:6200, misses:1979, evictions:1947
```

总分截图：

```
xjy1170500913@ubuntu:~/桌面/cachelab-handout$ python 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        1227
Trans perf 61x67          10.0        10        1979
          Total points    53.0        53
```

第 4 章 总结

4.1 请总结本次实验的收获

通过模拟缓存，对计算机的缓存结构有了更深的认识，理解了缓存命中、不命中的过程，了解了 LRU 的块驱逐机制等。
通过优化矩阵转置程序，深入理解了编写对缓存友好的程序的重要性。同时，对于不同的缓存结构、不同的数据结构，也要具体分析。

4.2 请给出对本次实验内容的建议

暂无。

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] 大卫 R.奥哈拉伦，兰德尔 E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社.2018.4