实验简介

Cache LAB 分为 Part A 和 B 两部分，这次实验的任务很明确，就是制作自己的缓存系统，具体来说是

- 实现一个缓存模拟器，根据给定的 trace 文件来输出对应的操作
- 利用缓存机制加速矩阵运算

我们需要修改的是 `csim.c`（Part A） 和 `trans.c`（Part B）。编译的时候只需要简单 `make clean` 和 `make`，然后就可以进行测试了。

文件说明

- csim.c：实现缓存模拟器的文件
- trans.c：实现矩阵转置的文件
- csim-ref：标准的缓存模拟器
- csim：由你实现的模拟器可执行程序
- tracegen：测试你的矩阵转置是否正确，并给出错误信息
- test-trans：测试你的矩阵转置优化的如何，并给出评分
- driver.py：自动进行测试评分

在每一次更新之后，首先用`make`生成文件，之后用相应的 test 跑分即可

前置说明

在向内存中写数据时，可能会发生以下几种情况：

- 写命中
- - write-through：直接写内存
    - write-back：先写Cache，当该行被替换时再写内存。此时需要一个额外的dirty位
- 写不命中
- - write-allocate：将内存数据读入Cache中，再写Cache
    - no-write-allocate：直接写内存

Cache 失效的三种原因：

- Cold miss：刚刚使用 Cache 时 Cache 为空，此时必然发生 Cache miss
- Capacity miss：程序最经常使用的那些数据(工作集,working set)超过了Cache的大小
- Conflict miss：Cache容量足够大，但是不同的数据映射到了同一组，从而造成Cache line反复被替换的现象

Cache 替换策略

Cache 工作原理要求它尽量保存最新数据

当从主存向 Cache 传送一个新块，而 Cache 中可用位置已被占满时，就会产生 Cache 替换的问题

常用的替换算法有下面三种
(1) LFU

LFU（Least Frequently Used，最不经常使用）算法将一段时间内被访问次数最少的那个块替换出去

每块设置一个计数器，从 0 开始计数，每访问一次，被访块的计数器就增 1

当需要替换时，将计数值最小的块换出，同时将所有块的计数器都清零

这种算法将计数周期限定在对这些特定块两次替换之间的间隔时间内，不能严格反映近期访问情况，新调入的块很容易被替换出去

(2) LRU

LRU（Least Recently Used，近期最少使用）算法是把 CPU 近期最少使用的块替换出去

这种替换方法需要随时记录Cache中各块的使用情况，以便确定哪个块是近期最少使用的块

每块也设置一个计数器，Cache 每命中一次，命中块计数器清零，其他各块计数器增 1；当需要替换时，将计数值最大的块换出

LRU 算法相对合理，但实现起来比较复杂，系统开销较大。这种算法保护了刚调入 Cache 的新数据块，具有较高的命中率

LRU 算法不能肯定调出去的块近期不会再被使用，所以这种替换算法不能算作最合理、最优秀的算法

但是研究表明，采用这种算法可使 Cache 的命中率达到 90% 左右。

(3) 随机替换

最简单的替换算法是随机替换

随机替换算法完全不管 Cache 的情况，简单地根据一个随机数选择一块替换出去

随机替换算法在硬件上容易实现，且速度也比前两种算法快；缺点则是降低了命中率和 Cache 工作效率

Part A: Writing a Cache Simulator

前置及要求

讲义上首先给我们提供了一个程序示例

```
valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

执行后我们可以看到输出如下：（输入的 trace 文件的内容）

```
.....
I  04e85fb6,5
 S ffefffbe8,8
I  04f18790,3
I  04f18793,7
 L 0520fe78,8
I  04f1879a,6
I  04f187a0,5
I  04f187a5,2
I  04f187c0,3
I  04f187c3,3
I  04f187c6,2
==20846==
==20846== Counted 0 calls to main()
==20846==
==20846== Jccs:
==20846==   total:         157,950
==20846==   taken:         66,750 (42%)
==20846==
```

```
==20846== Executed:
==20846==   SBs entered:   167,583
==20846==   SBs completed: 108,528
==20846==   guest instrs:  899,088
==20846==   IRStmts:       5,345,796
==20846==
==20846== Ratios:
==20846==   guest instrs : SB entered  = 53 : 10
==20846==        IRStmts : SB entered  = 318 : 10
==20846==        IRStmts : guest instr = 59 : 10
==20846==
==20846== Exit code:       0
```

trace文件中的指令具有如下形式：

```
I 0400d7d4,8
M 0421c7f0,4
L 04f6b868,8
S 7ff0005c8,8
```

即每行代表一个或两个内存访问。每行的格式是

```
[空格][操作类型][空格][内存地址][逗号][大小]
```

操作字段表示存储器访问的类型，其中：

```
I 表示指令加载
L 表示数据加载
S 表示数据存储
M 表示数据修改（即数据存储之后的数据加载）
```

每个 `I` 前面都没有空格。每个 `M`，`L` 和 `S` 之前总是有空格

地址字段指定一个 32 位的十六进制存储器地址

大小字段指定操作访问的字节数；

通俗地解释一下各种操作：

①对于 `I` 指定地操作，实验说明中提到，我们不需要考虑：

意思就是 valgrind 运行地时候第一个指令总是为操作 `I`

②对于 `L` 以及 `S` 指定的操作，我们简单地可以认为这两个操作都是对某一个地址寄存器进行访问（读取或者存入数据）

③对于 `M` 指定的操作，可以看作是对于同一地址连续进行 `L` 和 `S` 操作

然后实验给我们提供了一个程序 csim-ref，我们要做的就是写出一个和它功能一样的程序

```
root@lepPwn:~/CTF/study/csapp/cachelab-handout# ./csim-ref -h
Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
  -h         Print this help message.
  -v         Optional verbose flag.
  -s <num>   Number of set index bits.
  -E <num>   Number of lines per set.
  -b <num>   Number of block offset bits.
  -t <file>  Trace file.

Examples:
  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace
```

题解

因为没了解过这方面知识，所以按照逆序学习了，先用 IDA 逆向程序后抄了一份代码，然后分析内容

```c
#include "cachelab.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <getopt.h>
#include <errno.h>

typedef unsigned long long uint64_t;
typedef uint64_t mem_addr_t;
struct cache_line_t{
    mem_addr_t tag;          // 标记位
    int valid;               // 有效位
    unsigned int lru;        // 最近访问的次数
};
typedef struct cache_line_t* cache_set_t;
typedef cache_set_t* cache_t;

int verbosity;
int s;                       // 组索引位数量
int b;                       // Number of block offset bits（每行块数，块偏移位数量）
int E;                       // Number of lines per set（每个组的行数）
int miss_count;
int hit_count;
int eviction_count;
int func_counter;
int B;                       // 块大小（字节），这个没用到
int S;                       // Number of set index bits（组数）
uint64_t lru_counter = 1;
char* trace_file;            // Trace file（trace 文件）
cache_t cache;
mem_addr_t set_index_mask;
trans_func_t func_list[100];


void initCache(){
```

```
    cache_set_t* v0;
    int i;
    int j;

    /* *
     * 由此可见 Cache 的结构是一个二维数组
     * 首先分配规定的组数 S，之后在每个组里分配规定的行数 E
     * 将每个块上的结构体内部变量的值都清零
     * */
    cache = (cache_t)malloc(8 * S);
    for(i = 0; i < S; ++i){
        v0 = &cache[i];
        *v0 = (cache_set_t)malloc(24 * E);
        for(j = 0; j < E; ++j){
            cache[i][j].valid = 0;
            cache[i][j].tag = 0;
            cache[i][j].lru = 0;
        }
    }
    /* *
     * 源代码如下：
     * double v1;
     * uint64_t v2;
     * v1 = pow(2.0, (double)s) - 1.0;
     * if(v1 >= 9.223372036854776e18)
     *     v2 = (unsigned int)(signed int)(v1 - 9.223372036854776e18) ^
0x8000000000000000;
     * else
     *     v2 = (unsigned int)(signed int)v1;
     * set_index_mask = v2;
     * */
    set_index_mask = (1 << s) - 1;
}


void freeCache(){
    int i;

    for(i = 0; i < S; ++i)
        free(cache[i]);     // 释放行数部分的堆块，在 heap 里的变量
    free(cache);            // 释放组数部分的堆块，在 bss 段上的全局变量
}


void accessData(mem_addr_t addr){
    int i;
    int ia;
    unsigned int eviction_line = 0;
    uint64_t eviction_lru = -1;
    /* *
     * 从 address 取出 offset -> offset = converse(offset, b);
     * for (int i = 0; i < b; ++i) {
     *     offset = offset * 2 + address % 2;
     *     address >>= 1;
     * }
     *
     * 从 address 取出 tag -> setindex = converse(setindex, s);
     * for (int i = 0; i < s; ++i) {
```

```
     *     setindex = setindex * 2 + address % 2;
     *     address >>= 1;
     * }
     * */
    mem_addr_t tag = addr >> ((unsigned char)s + (unsigned char)b);  // tag = addr >> b >> s ;
    cache_set_t cache_set = cache[(addr >> b) & set_index_mask];

    for(i = 0; ; ++i){
        /*  遍历完 cache_set 后如果还是找不到所需的数据 */
        if(i >= E){
            ++miss_count;  // 未命中数 +1
            if(verbosity)
                printf("miss ");
            /* 查找谁适合被删除（LRU） */
            for(ia = 0; ia < E; ++ia){
                if(cache_set[ia].lru < eviction_lru){
                    eviction_line = ia;
                    eviction_lru = cache_set[ia].lru;
                }
            }
            /* 如果当前要被删除的块内已经有数据了 */
            if(cache_set[eviction_line].valid){
                ++eviction_count;  // 删除数 +1
                if(verbosity)
                    printf("eviction ");
            }
            /* 写入或覆盖这个作为被删除的 cache_line 里 */
            cache_set[eviction_line].valid = 1;
            cache_set[eviction_line].tag = tag;
            cache_set[eviction_line].lru = lru_counter++;  // 使用数 +1
            return;
        }
        if(cache_set[i].tag == tag && cache_set[i].valid)
            break;
    }
    ++hit_count;  // 碰撞数 +1
    if(verbosity)
        printf("hit ");
    cache_set[i].lru = lru_counter++;  // 使用数 +1
}


void replayTrace(char* trace_fn){
    unsigned int len;
    mem_addr_t addr;
    FILE* trace_fp;
    char buf[1000];

    addr = 0;
    len = 0;
    trace_fp = fopen(trace_fn, "r");
    /* 检测将要打开的文件是否存在 */
    if(!trace_fp){
        int* v1 = __errno_location();
        char* v2 = strerror(*v1);
        fprintf(stderr, "%s: %s\n", trace_fn, v2);
        exit(1);
```

```
152     }
153     while(fgets(buf, 1000, trace_fp)){
154         if(buf[1] == 'S' || buf[1] == 'L' || buf[1] == 'M'){
155             sscanf(&buf[3], "%llx,%u", &addr, &len);  // 如，读入: L 10,1
156             if(verbosity)
157                 printf("%c %llx,%u ", (unsigned int)buf[1], addr, len);
158             accessData(addr);
159             if(buf[1] == 'M')                           // 如果当前指令是修改指令，则上行的 accessData
160                 accessData(addr);                       // 用来读取数据，这行的 accessData 用来写入数据
161             if(verbosity)
162                 putchar('\n');
163         }
164     }
165     fclose(trace_fp);
166 }
167 
168 
169 void printUsage(char** argv){
170     printf("Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n", *argv);
171     puts("Options:");
172     puts("  -h         Print this help message.");
173     puts("  -v         Optional verbose flag.");
174     puts("  -s <num>   Number of set index bits.");
175     puts("  -E <num>   Number of lines per set.");
176     puts("  -b <num>   Number of block offset bits.");
177     puts("  -t <file>  Trace file.");
178     puts("\nExamples:");
179     printf("  linux>  %s -s 4 -E 1 -b 4 -t traces/yi.trace\n", *argv);
180     printf("  linux>  %s -v -s 8 -E 2 -b 4 -t traces/yi.trace\n", *argv);
181     exit(0);
182 }
183 
184 
185 int main(int argc, const char** argv, const char** envp){
186     char choice;
187 
188     while((choice = getopt(argc, (char* const*)argv, "s:E:b:t:vh")) != -1){
189         switch(choice){
190         case 'E':
191             E = atoi(optarg);  // optarg 是 getopt 函数内带有的变量
192             break;
193         case 'b':
194             b = atoi(optarg);
195             break;
196         case 'h':
197             printUsage((char**)argv);
198             return 0;
199         case 's':
200             s = atoi(optarg);
201             break;
202         case 't':
203             trace_file = optarg;
204             break;
205         case 'v':
```

```
            verbosity = 1;
            break;
        default:
            printUsage((char**)argv);
            return 0;
        }
    }
    if(!s || !E || !b || !trace_file){
        printf("%s: Missing required command line argument\n", *argv);
        printUsage((char**)argv);
    }
    S = 1 << s;  // 用组索引位数量算出组数，原式: S = (signed int)pow(2.0, (double)s);
    B = 1 << b;  // 用块偏移位数量算出块的大小，原式: B = (signed int)pow(2.0, (double)b);
    initCache();
    replayTrace(trace_file);
    freeCache();
    printSummary(hit_count, miss_count, eviction_count);  // 打印最后的结果
    return 0;
}
```

Part B：Optimizing Matrix Transpose

优化先跳了。。

参考链接

https://blog.csdn.net/xbb224007/article/details/81103995

https://blog.csdn.net/zjwreal/article/details/80926046

https://blog.csdn.net/yhb1047818384/article/details/79604976