华中科技大学

计算机系统结构实验报告

姓　　名：　　练炳诚

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS1705

学　　号：　　U201714707

指导教师：　　万继光

分数	
教师签名	

2020 年.　4 月.　28 日

目　录

1. Cache 模拟器实验

1.1. 实验目的

加深 Cache 缓存组成结构对 C 程序性能的影响的理解。

1.2. 实验环境

VMware 虚拟机环境下：

Ubuntu version 16.04.12

Linux version 4.15(amd64)

Gcc version 5.4.0

1.3. 实验思路

1.3.1 相关数据结构定义

实验要求是实现一个 Cache 模拟器，分析测试文件发现，不需要 Cache 槽中存放实际数据，故设计 Cache 槽如下：

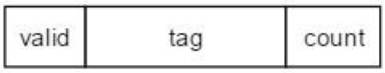

图 1.1 Cache 槽设计

其中，valid 是有效位，标明该 Cache 槽中数据是否有效；tag 字段是标志位，用于唯一标识数据；count 是 LRU 算法计数器，每次访问访问 Cache 时，若该槽未命中，则对 count 进行加 1，若命中,则 count 清零。在进行替换时，选择 count 最大的 Cache 槽进行替换。

C 语言定义如下:

```
/* cache struct define here */
typedef struct cache_line{
    char valid;
    unsigned long long tag;
    unsigned long long count;
}cache_line;
```

图 1.2 Cache 槽 C 语言定义

一些相关的变量定义如下：

```
/* global paramater define here */
// command line paramaters
unsigned int s = 0, E = 0, b = 0, v = 0;
// cache file address
char* t = NULL;
// counters
int miss_count = 0, hit_count = 0, eviction_count = 0;
// num of index and address
unsigned group_num = 0;
unsigned addr_num = 0;
// cache
cache_line** cache = NULL;
// addr mask
unsigned long long mask = 0;
```

图 1.3 相关变量定义

其中，s、E、b、v、t 是命令行参数，s、E、b 分别为组索引位数、组内块数、内存块地址位数，v 为 1 表示打印轨迹信息，为 0 表示不打印(默认)，t 指向测试文件的绝对路径，miss_count、hit_count、eviction_count 分别为缺失次数、命中次数、替换次数，group_num 为 Cache 组数，计算公式为 group_num=2^s，addr_num 为内存块数，计算公式为 addr_num=2^b，cache 是一个二重指针，指向一个二维 cache_line 数组，数组每一行为一个 cache 组，每行有 E 列，共有 group_num 行，mask 是掩码，用来获取 cache 查找时的 tag 字段。

1.3.2 模拟器总体结构实现

main 函数中接受命令行参数，通过对 arv[]数组中的参数进行分析，判定输入参数的完整性、合法性，若参数不完整或者非法，则打印帮助信息（输入的命令行参数至少包含 s、E、b，且 s、b 非负，E 大于 0）。在完成参数检验后，初始化 cache、运行 cache、模拟访问 cache，销毁 cache，最后调用 printSummary 函数打印结果信息。流程图如图 1.4 所示。

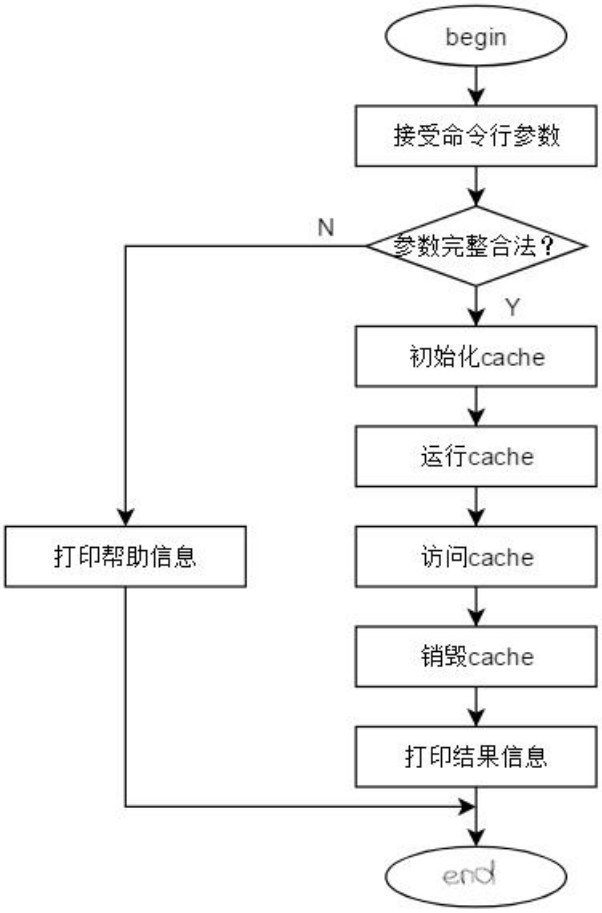

图 1.4 mian 函数流程图

1.3.3 Cache 初始化及销毁实现

初始化 Cache 时，调用 malloc 函数动态生成一个二维 cache_line 的结构数组，即 cache 指针指向一个 cache_line 指针数组，每个数组元素指向一个 cache_line 数组，每行作为一个 cache 组，共 group_num 行，每行有 E 列。对初始 cache 的 valid 全部置为 0，表示数据无效，tag 字段、count 字段全部置为 0。

销毁 Cache 时，调用 free 函数按行进行销毁，先对 cache_line 指针数组指向的空间进行 free,最后 free 掉指针数组空间。

1.3.4 运行 Cache 函数 runCache 实现

函数接受的参数是一个字符串指针，指向测试文件的绝对路径，runCache 首先调用 fopen 函数以只读方式打开该文件，若打开失败则打印错误信息并退出，否则，调用 fscanf 格式化输入函数，将测试文件按行读取，并将读取到的操作（I、L、M、S）、地址、大小分别赋给变量 op、addr、size，由于 I 表示指令装载，而

实验只关心数据 cache，故对 op 为 I 的行进行忽略；如果 op 不是 I，且 v 为 1，则需打印踪迹信息，即打印出”op addr,size”；对 op 为 L、S 的操作，根据 addr 访问一次 Cache，即调用 accessCache 函数；对 op 为 M 的操作，可以视为一次 L 操作和一次 M 操作，故需调用两次 accessCache 函数。当读取完毕后，关闭文件。实现如下：

```
void runCache(char* t){
        unsigned long long addr;
        unsigned int size;
        char op[1];
        FILE* fp = NULL;
        if((fp = fopen(t, "r")) == NULL){
                printf("open trace file error.\n");
                exit(-1);
        }
        while(fscanf(fp, " %c %llx,%d", op, &addr, &size) > 0){
                if(v == 1 && op[0] !='I'){
                        printf("%c %llx,%d ", op[0], addr, size);
                }
                switch(op[0]){
                case 'L' :
                case 'S' :
                        accessCache(addr);
                        break;
                case 'M' :
                        accessCache(addr);
                        accessCache(addr);
                        break;
                default :
                        break;
                }
                if(v == 1 && op[0] != 'I'){
                        putchar('\n');
                }
        }
        fclose(fp);
```

图 1.5 runCache 代码实现

1.3.5 访问 Cache 函数 accessCache 实现

内存地址的结构划分如下:

图 1.6 内存地址划分

函数 accessCache 接受参数 addr，将 addr 右移 b 位得到相应的 cache 组号即组索引 set_inde=addr>>b，取 cache 组 cache_group=cache[set_index]，addr 右移(s+b)位得到相应的 tag 字段，即 tag=addr>>(s+b)。

进行 Cache 搜索时，首先对 cache 组中所有槽的计数器 count 加 1，然后开始查找。当 tag 字段相等并且 valid 字段为 1 时表示命中，若 v 为 1 则打印“hit”，将 count 清零，hit_count 加 1，返回上层调用；否则，表示未命中，miss_count 加 1，此时再次进行搜索，记录搜索过程中的最大计数值 max_count 及相应下标

max_index，若所有的 Cache 槽中 valid 字段都是 1，则表示 cache 组已满，此时对计数值最大的 cache 槽进行替换，eviction_count 加 1，若 v 为 1 则打印”eviction”，表示替换；否则，选择第一个 valid 为 0 的 cache 槽，填入 tag 字段，标记 valid 为 1，并将计数器清零。

accessCache 的流程如图 1.7 所示。

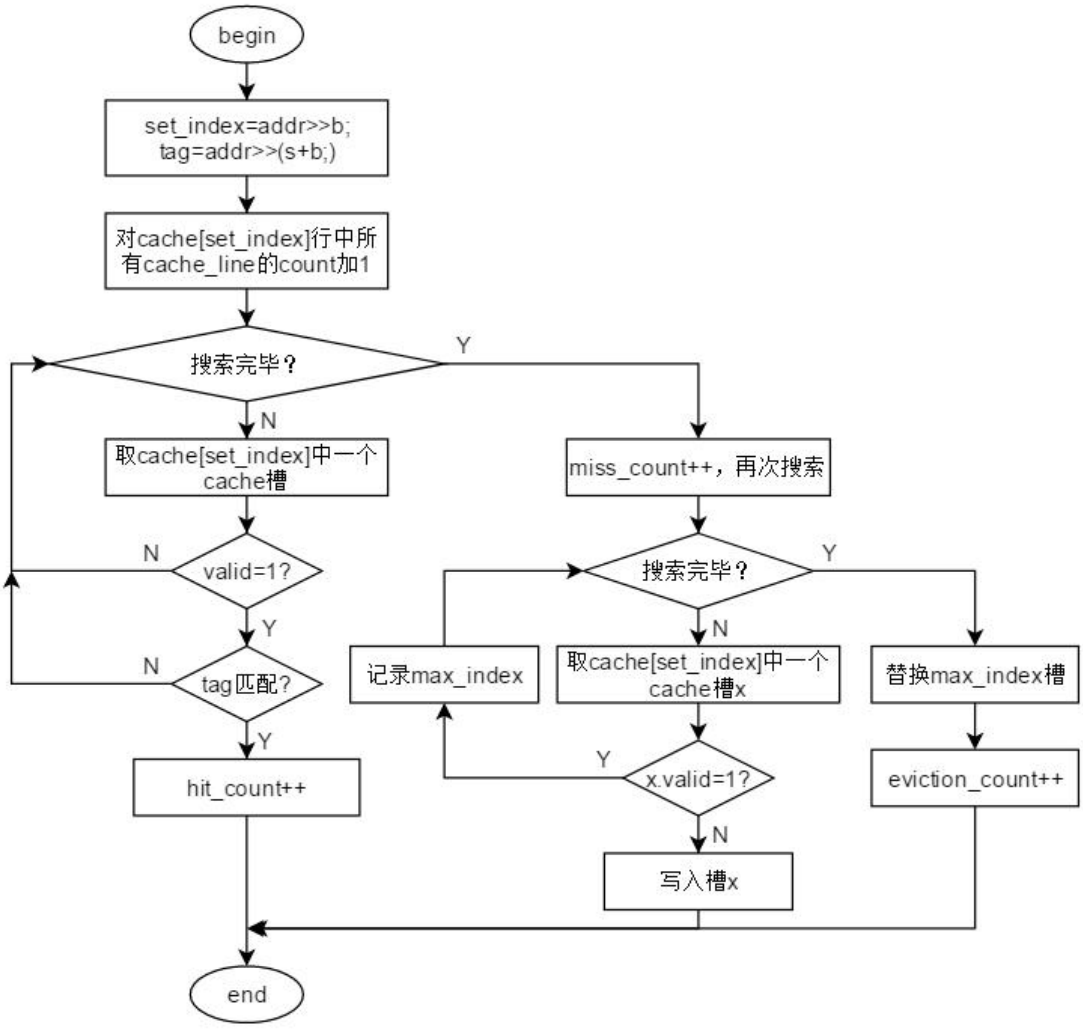

图 1.7 accessCache 函数流程图

1.4. 实验结果和分析

1.4.1 程序功能性分析

首先测试-h 参数打印帮助信息，如图 1.8 所示。

```
root@ubuntu:/usr/local/src/exp1/cachelab-handout# ./csim -h
Usage: ./csim [-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 -s 4 -E 1 -b 4 -t traces/yi.trace
  linux>  ./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace
root@ubuntu:/usr/local/src/exp1/cachelab-handout#
```

图 1.8 -h 参数测试

打印信息显示了 csim 程序所需的各参数及参数类型，并给出了使用样例。

测试参数检查功能，当缺失 s、E、b 中的某一项参数时，或者参数非法时，均打印上述帮助信息，说明参数验证功能正常，此处不再展示。

按照样例 1 给出的参数测试 yi.trace 文件，结果如图 1.9 所示。

```
root@ubuntu:/usr/local/src/exp1/cachelab-handout# ./csim -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
root@ubuntu:/usr/local/src/exp1/cachelab-handout# ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
root@ubuntu:/usr/local/src/exp1/cachelab-handout#
```

图 1.9 -样例 1 测试结果

结果显示与参考 Cache 模拟器的输出一致。更改一个不存在的文件路径 traces/null.trace，结果如图 1.10 所示。

```
root@ubuntu:/usr/local/src/exp1/cachelab-handout# ./csim -s 4 -E 1 -b 4 -t traces/null.trace
open trace file error.
root@ubuntu:/usr/local/src/exp1/cachelab-handout#
```

图 1.10 错误路径测试结果

结果显示对错误路径的文件程序报错。

运行 test-csim 测试程序，对 csim 和参考 cache 程序 csim-ref 比较进行功能测试，结果如图 1.11 所示。

```
root@ubuntu:/usr/local/src/exp1/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
root@ubuntu:/usr/local/src/exp1/cachelab-handout#
```

图 1.11 cism 功能测试结果

结果显示 csim 功能正确。

1.4.2 探究 Cache 组成结构对性能的影响

通过以下过程探究 Cache 组索引位数(s)、内存块地址位数(b)和 Cache 的关联度(E)对程序性能的影响，测试结果见表 1.1。

表 1.1 不同参数配置对程序性能的影响

s	b	E	测试文件/行数	结果	说明
0	0	4	yi.trace/7	hits:2 misses:7 evictions:3	该情况实际是全相联 Cache，共 4 行，每行 1 个字，由于只能装载 4 个字，故发生了 3 次替换
0	0	8	yi.trace/7	hits:2 misses:7 evictions:0	同上，但此时 Cache 能装载至少 8 个字，故不会发生替换，即 eviction=0
2	2	4	long.trace/267988	hits:250081 misses:36883 evictions:36867	此时为 4 路组相联，共 4 组，每组 4 行，每行装载 4 个字
2	2	8	long.trace/267988	hits:253153 misses:33811 evictions:33779	组内行数翻倍后，命中率有所提升
3	2	4	long.trace/267988	hits:253665 misses:33299 evictions:33267	组数翻倍后，带来的命中率提升略小于增加组内行数，但替换次数略小于前者
2	3	4	long.trace/267988	hits:262374 misses:24590 evictions:24574	将行装载字数翻倍后，命中率提升高于增加组内行数及组数，且显著降低了替换率
4	4	16	long.trace/267988	hits:278763 misses:8201 evictions:7945	三个参数均增加时，显著提升 Cache 命中率，降低替换次数，提高程序性能

结果表明，主存地址空间一定时，通常情况下，当 Cache 的组数越多、组内

行数越大、每行装载的数据块越多时，Cache 命中率越高，即程序性能越好，但由于 Cache 容量越大，造假越昂贵，故通常会在容量和价格间做一个折中，即选取一定的 s、E、b 比例，使得对大对数程序来说能够接近一个最大的命中率。

当组数和行装载字数一定时，提升组内行数可以降低冲突率（组内全相联），进而降低替换次数，提高一定的命中率；当行装载字数一定时，增加 cache 组数提升的效果相比增加组内行数命中率稍低，但替换次数降低了，原因是增加组数使得总体冲突率降低，而由于两种情况下 Cache 容量一定，故关联度高的由于组内行数相对较小故更容易发生替换，但命中率相对较高，体现了全相联到直接映射间转换的过程。

2. 总结和体会

通过本次实验，在代码层面模拟实现了组相联 Cache 的实现，通过 C 调整 Cache 组索引位数 s（即 Cache 组数）、内存块地址位数 b（即 Cache 槽包含的字数）和 Cache 的关联度 E（即 Cache 组内快数）来研究 Cache 组成结构对程序性能的影响，经过对比分析得出 Cache 组成结构对程序性能影响的初步结论。

相对硬件实现 Cache，代码的方式更加简单，而且可以通过调整参数的形式很方便地修改 Cache 的组成结构（如 s=0,b=4,E=x 表示有 x 行，每行 4 个字的全相联 Cache）。

3. 对实验课程的建议

本课程的实验相对较为简单，能够顺利完成，但是在上学期的组成原理实验中已经通过电路仿真实现过组相联 cache 及 LRU 淘汰算法，相对代码实现的 cache 更加底层，因此个人认为该 cache 实验对个人的提升不大，建议可以建立关于互联网络或者并行计算的相关实验题目，这些是之前的课程很少接触的内容。