# 华中科技大学

# 计算机系统结构实验报告


姓　　名：　　熊逸钦

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS1701

学　　号：　　U201714501

指导教师：　　童薇、施展


| 分数 | |
|---|---|
| 教师签名 | |

2020 年 4 月 18 日

# 目 录

# 1. Cache 模拟器实验

本次实验中需要通过编写一个 200-300 行的 C 程序来模拟 Cache 缓存的行为。

给出内存访问的轨迹，要求编写的 Cache 模拟器程序能够模拟缓存相对内存访问轨迹的命中/缺失行为。并最终输出该内存访问轨迹的命中、缺失和（缓存行）淘汰/驱逐的总数。

## 1.1. 实验目的

（1）理解 Cache 的工作原理；

（2）掌握实现一个高效模拟器的方法。

## 1.2. 实验环境

本次实验对环境的要求不高，只需要一个基础的 C 语言程序的编译运行环境以及 64 位的 Linux 操作系统即可。

实验中我的具体实验环境如下：

（1）操作系统：Ubuntu 18.04 LTS x64

（2）开发语言：C

（3）编译器：gcc 7.4.0

（4）编辑器：Visual Studio Code

## 1.3. 实验思路

（1）分析需求

程序的输入是内存访问轨迹，以测试用例中的 yi.trace 为例，其访存轨迹如下：

```
L 10,1
M 20,1
L 22,1
S 18,1
L 110,1
L 210,1
M 12,1
```

本次实验只考虑数据 Cache 的性能，对指令 Cache 不进行考虑，因此不考虑行首字符为 I 的指令访存轨迹。对于上面的例子，其中每一行的第一个字符为空格，第二个字符的取值集合为{L, S, M}，代表操作类型 operation。其中 L 表示 Load 读取内存，S 表示 Store 写入内存，M 表示 Modify 修改内存（可以看作是对于同一个地址先 Load 再 Store）。后面以逗号隔开的两个数字分别是访存地址 address（十六进制）、访问的内存字节数量 size（十进制）。由于本次实验假设内存访问的地址总是正确对齐的，即一次内存访问从不跨越块的边界，因此可忽略访问轨迹中给出的访问请求大小。

实验规定程序执行时的命令行格式如下：

```
csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

其中：

```
-h: 显示帮助信息（可选）
-v: 显示轨迹信息（可选）
-s <s>: 组索引位数
-E <E>: 关联度（每组包含的缓存行数）
-b <b>: 内存块内地址位数
-t <tracefile>: 内存访问轨迹文件名
```

根据执行模拟器程序时携带的命令行参数建立 Cache 后，逐行读入并分析操作类型和访存地址，对于 L 和 S 操作，若命中 Cache 则将 LRU 计数重置，否则查找 Cache 空行进行加载，若无空行则需要根据各行 LRU 计数的值找出被淘汰行实现替换。对于 M 操作，可以看作对同一地址依次执行一次 L 和 S 操作。

（2）理解代码框架

实验中需要完成的是 csim.c 文件，已给出代码框架，其中包含 cache 行的结构体定义，包含有效位、标识位和 LRU 计数。此外还有关键全局变量如下：

```
/* Globals set by command line args */
int verbosity = 0; /* 若为 1 则显示详细命中信息 */
int s = 0;              /* 组号位数 */
int b = 0;              /* 块内偏移位数 */
int E = 0;              /* 组内行数*/

/* Derived from command line args */
int S; /* 组数 */
int B; /* 块大小 (字节) */

/* Counters used to record cache statistics */
int miss_count = 0; /* 缺失计数 */
int hit_count = 0;   /* 命中计数 */
int eviction_count = 0; /* 驱逐计数 */
unsigned long long int lru_counter = 1; /* LRU 计数 */
```

程序框架里已经给出的函数中，initCache()使用 malloc 函数根据命令行参数的值来动态申请 Cache 所需的内存空间；printUsage()展示程序用法；main()获取命令行参数，调用相关函数进行模拟。

可以看到 Cache 为一个二维结构，Cache 指针是 S 个 Cache 块头指针构成的数组的头指针，每个 Cache 块头指针为 E 个 Cache 行结构体组成的数组的起始地址。Cache 的替换算法采用 LRU 算法，LRU 计数器的初始值为 unsigned long long 类型的最大值，每次命中都将计数器重置为初始值，其他情况下计数器值递减 1。

程序框架里需要完善的函数中，freeCache()释放 initCache 函数所申请的内存空间；accessData(addr)对地址 addr 在 Cache 中查找，对命中、缺失等情况作出相应处理并统计信息；replayTrace()读取内存访问轨迹文件内容，从每一行解析出操作类型、目标内存地址，若为 L 或 S 操作则调用一次 accessData 函数，若为 M 操作则对于同一地址连续调用两次 accessData 函数。

（3）编写程序

根据上述（2）中的理解和分析，完善 freeCache 函数、accessData 函数和 replayTrace 函数。

1. freeCache 函数

由于 Cache 是一个二维结构，一个 Cache 包含 S 个组，每组中有 E 个 Cache 行，因此在使用 free 函数释放内存时首先以组为单位，对 Cache 每一组进行 free 操作，然后再对 Cache 进行 free 操作，最后将 Cache 指针置为 NULL。

完成后该函数代码如下：

```
/*
 * freeCache - free allocated memory
 */
void freeCache()
{
    int i;
    for (i = 0; i < S; i++)
        free(cache[i]);
    free(cache);
    cache = NULL;
}
```

2. accessData 函数

首先使用位运算从接收参数 addr 中提取出组号 set_index 和标识 tag。

然后对于 set_index 指定的组中的 E 个 Cache 行进行遍历。对于有效位为 1 的每一行，首先将 LRU 计数-1（LRU 计数的默认值为 unsigned long long 类型的最大值），然后维护当前组中 LRU 最小值对应的 Cache 行号，若发生驱逐，则作为被驱逐行的行号，最后若目标 tag 与当前 tag 匹配，说明命中，记录下命中

行号。对于有效位为 0 的行，记录下第一次出现的有效位为 0 的行，作为发生数据载入时的目标行（空行），此后跳过有效位为 0 的行。

再对命中情况作出处理，若命中行号不为-1，说明 Cache 命中，则打印命中信息、增加命中计数、将命中行 LRU 计数重置、将命中行号重置为-1。

最后对缺失情况作出处理，首先增加缺失计数。若空闲行号不为-1，说明存在空行，则打印载入信息、将空闲行信息改写为目标行信息、重置空闲行号为-1。否则说明不存在空行，则对被驱逐行号对应的行进行驱逐：增加驱逐计数、把目标行信息覆盖到被驱逐行，最后重置最小 LRU 计数为最大值。

完成后该函数代码如下：

```
/*
 * accessData - Access data at memory address addr.
 *     If it is already in cache, increast hit_count
 *     If it is not in cache, bring it in cache, increase miss count.
 *     Also increase eviction_count if a line is evicted.
 */
void accessData(mem_addr_t addr)
{
    int i;
    lru_counter = ULONG_MAX;
    unsigned int eviction_line = 0;
    mem_addr_t set_index = (addr >> b) & set_index_mask;
    mem_addr_t tag = addr >> (s + b);

    cache_set_t cache_set = cache[set_index];
    int hit_index = -1;        //命中行号
    int invalid_index = -1; //空闲行号
    /*  在 set 中的每一行查找  */
    for (i = 0; i < E; i++)
    {
        if (cache_set[i].valid)
        {
            cache_set[i].lru--; //无论是否命中，默认将 LRU 计数-1
            if (cache_set[i].lru < lru_counter)
            {
                lru_counter = cache_set[i].lru;
                eviction_line = i; //找出 set 中 LRU 计数最小值，若须替换则为替换行
            }
            if (cache_set[i].tag == tag)
            {
                /*  命中  */
                hit_index = i;
```

```
                    }
                }
                else if (invalid_index == -1)
                    invalid_index = i; //如果有的话，记录下第一个空行
            }
            /* 命中处理 */
            if (hit_index != -1)
            {
                if (verbosity)
                    printf("Hit! line:%d tag:%lld\n", hit_index,
cache_set[hit_index].tag);
                hit_count++;
                cache_set[hit_index].lru = ULONG_MAX; //LRU 计数重置为最
大值
                hit_index = -1;  //重置命中行号为-1
            }
            /* 缺失处理 */
            else
            {
                miss_count++;
                /* 尝试找空位填入 */
                if (invalid_index != -1)
                {
                    if (verbosity)
                        printf("Miss_Insert! line:%d tag:%lld\n", invalid_index,
cache_set[invalid_index].tag);
                    cache_set[invalid_index].tag = tag;
                    cache_set[invalid_index].lru = ULONG_MAX;
                    cache_set[invalid_index].valid = 1;
                    invalid_index = -1; //重置空闲行号为-1
                }
                /* 没有空位则替换 */
                else
                {
                    if (verbosity)
                        printf("Miss_Evict! line:%d tag:%lld\n", eviction_line,
cache_set[eviction_line].tag);
                    eviction_count++;
                    cache_set[eviction_line].tag = tag;
                    cache_set[eviction_line].lru = ULONG_MAX;
                    lru_counter = ULONG_MAX; //重置最小 LRU 计数
                }
            }
        }
```

3. replayTrace 函数

在 replayTrace 函数中，将文件中满足首字符为空格的内容按行读取到 buf 中，同时读 addr 和 len 参数。从 buf[0]获取操作类型，若为 L 或 S 操作则调用一次 accessData 函数，若为 M 操作则对同一地址连续调用两次 accessData 函数。

完成后该函数代码如下：

```
/*
 * replayTrace - replays the given trace file against the cache
 */
void replayTrace(char *trace_fn)
{
    char buf[1000];
    mem_addr_t addr = 0;
    unsigned int len = 0;
    FILE *trace_fp = fopen(trace_fn, "r");

    /* 跳过空格，读取行内容 */
    while (fscanf(trace_fp, " %s %llx,%u", buf, &addr, &len) != EOF)
    {
        if (verbosity)
            printf("%s %llx,%u\n", buf, addr, len);
        switch (buf[0])
        {
        case 'L':
            /* Load */
            accessData(addr);
            break;
        case 'S':
            /* Store */
            accessData(addr);
            break;
        case 'M':
            /* Modify */
            accessData(addr);
            accessData(addr);
            break;
        default:
            break;
        }
    }

    fclose(trace_fp);
}
```

## 1.4. 实验结果和分析

完成代码之后在源码所在根目录打开终端，使用老师提供的已经编写好的 MakeFile 进行编译。首先输入 make clean 命令清除上一次编译生成的文件，然后输入 make 命令进行编译，并且自动将源代码打包为 tar 压缩文件，如下图 1.1 所示。

```
yiqin0411@x1carbon:~/文档/CA_Lab/cachelab-handout$ make clean
rm -rf *.o
rm -f *.tar
rm -f csim
rm -f test-trans tracegen
rm -f trace.all trace.f*
rm -f .csim_results .marker
yiqin0411@x1carbon:~/文档/CA_Lab/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf yiqin0411-handin.tar  csim.c trans.c
csim.c
trans.c
```

图 1.1 生成目标程序

编译完成后就可以使用老师提供的 test-csim 测试程序对程序正确性进行验证，验证结果如下图 1.2 所示：

```
yiqin0411@x1carbon:~/文档/CA_Lab/cachelab-handout$ chmod +x test-csim
yiqin0411@x1carbon:~/文档/CA_Lab/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
```

图 1.2 测试程序运行结果

从上述结果可知，本次实验中的 Cache 模拟器的模拟行为与参考模拟器的行为一致，正确实现了 Cache 的模拟功能。

再使用带-v 参数的命令行对模拟器行为进行分析，选用 yi.trace 作为用例，构造一个 4 组、每组 2 行，块内偏移位数为 4 的 Cache。

首先使用老师提供的手工演算 Excel 工具预测程序行为，以 yi.trace 为例的手工演算过程如下图 1.3 所示：

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| L | 10 | 1 | 16 | 1 | 10 | 0 | 1 | M |
| M | 20 | 1 | 32 | 2 | 4 | 0 | 2 | M,H |
| L | 22 | 1 | 34 | 2 | 6 | 0 | 2 | H |
| S | 18 | 1 | 24 | 1 | 2 | 0 | 1 | H |
| L | 110 | 1 | 272 | 17 | 14 | 1 | 1 | M |
| L | 210 | 1 | 528 | 33 | 2 | 2 | 1 | M,E |
| M | 12 | 1 | 18 | 1 | 12 | 0 | 1 | M,E,H |
| | | | 十进制数 | 块号 | 块内偏移 | #Tag | #S | |

图 1.3 手工演算的程序行为

从上图可以看到，第一步 L 操作访问第 1 块，此时 Cache 为空故缺失，将第 1 块装入 Cache 第 1 组第 0 行。第二步 M 操作访问第 2 块，先缺失，将第 2 块装入 Cache 第 2 组第 0 行，然后下一次访问命中。第三步 L 操作访问第 2 块，命中。第四步 S 操作访问第 1 块，命中。第五步 L 操作访问第 17 块，缺失，将第 17 块装入 Cache 第 1 组第 1 行。第六步 L 操作访问第 33 块，缺失，将第 33 块装入 Cache 第 1 组，但相联数为 2，每组只有 2 行，Cache 第 1 组已满，根据 LRU 策略，应该淘汰第 1 组第 0 行。第七步 M 操作访问第 1 块，先缺失，将第 1 块装入 Cache，但 Cache 第 1 组已满，根据 LRU 策略，应该淘汰第 1 组第 1 行，然后下一次访问命中。

使用如下命令进行实际操作：

```
./csim -v -s 4 -E 2 -b 4 -t traces/yi.trace
```

实际过程如下图 1.4 所示。

```
yiqin0411@x1carbon:~/文档/CA_Lab/cachelab-handout$ ./csim -v -s 4 -E 2 -b 4 -t traces/yi.trace
L 10,1
Miss_Insert! line:0 tag:0
M 20,1
Miss_Insert! line:0 tag:0
Hit! line:0 tag:0
L 22,1
Hit! line:0 tag:0
S 18,1
Hit! line:0 tag:0
L 110,1
Miss_Insert! line:1 tag:0
L 210,1
Miss_Evict! line:0 tag:0
M 12,1
Miss_Evict! line:1 tag:1
Hit! line:1 tag:0
hits:4 misses:5 evictions:2
```

图 1.4 实际程序行为

# 2. 总结和体会

在计算机系统中对存储器的访问是非常频繁的操作，但 CPU 的速度和主存的速度相差几个量级，根据加快经常性事件的原理和程序的时间、空间局部性原理，为了提高整个系统的速度，在 CPU 和主存之间设置了 Cache，将经常使用的

数据存放在容量小但速度快的 Cache 中。在本次实验中，使用 C 语言实现了一个自定义规格 Cache 的模拟程序，模拟了 Cache 的构造和处理命中、缺失和冲突的过程。实验的代码框架中主要需要完成的就是访问 Cache 的程序段。在组内遍历 Cache 行，当有效位为 1 且 tag 匹配时命中，否则不命中，不命中时还需要区分载入新行还是替换已有行。

通过完成本次实验，整个过程使我对 Cache 的结构、Cache 命中和缺失时的行为、Cache 的 LRU 替换算法的规则有了更加深刻的理解和认识，实际的模拟器编写过程也让我对书本上的 Cache 模型有了实践层面的理解。

实验的指导书、代码框架、测试样例、手工演算工具，甚至实验环境的搭建教程都由老师提供完备，实验当天老师还开了直播进行实操讲解，真的非常感谢老师们的精心准备和耐心指导！

# 3. 对实验课程的建议

实验本身的内容很不错，就是只有一个实验感觉内容有点单薄。我个人感觉系统结构课程里面的互联网络这一部分内容有点抽象，或许可以增加一点实验课时，模拟一下互联网络，加深对概念的理解。