

**计算机系统结构实验报告**

|  |  |
| --- | --- |
| 姓 名： | 周大伟 |
| 学 院： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 班 级： | 校交1803 |
| 学 号： | U201815553 |
| 指导教师： | 万继光 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2021 年. 4月. 28日

**目 录**

[1. Cache模拟器实验 3](#_Toc13985)

[1.1. 实验目的 3](#_Toc20693)

[1.2. 实验环境 3](#_Toc17135)

[1.3. 实验思路 3](#_Toc626)

[1.4. 实验结果和分析 7](#_Toc15016)

[2. 总结和体会 8](#_Toc17721)

[3. 对实验课程的建议 8](#_Toc1438)

# Cache模拟器实验

## 实验目的

· 理解cache工作原理；

· 如何实现一个高效的模拟器

## 实验环境

Linux 64-bit ，C语言

## 实验思路

### 1.3.1 cache模拟

建立cache的数据结构，包括有效位valid、标签位tag和使用记录器lru。实现代码如下：

typedef struct cache\_line {

char valid;

mem\_addr\_t tag;

unsigned long long int lru;

} cache\_line\_t;

定义cache组：用数组模拟cache组，即建立关于cache组的一个指针，然后动态开辟所需要cache的数量。实现代码如下：

typedef cache\_line\_t\* cache\_set\_t;

定义cache：用cache\_set数组模拟cache，即一个cache包含多个cache组，同样先创建一个cache指针，然后根据所需要cache组的大小动态开辟cache的实际大小。实现代码如下：

typedef cache\_set\_t\* cache\_t;

### 1.3.2输入输出参数

verbosity：显示轨迹信息，输入指令中含-v则verbosity置1，表示要进行轨迹信息输出，否则verbosity置0。

s：组索引位数，输入指令中-s后面的数据为组索引位数，用S=2s表示所需要的cache组数。

b：内存块内地址位数，输入指令-b后面的数据为块内存地址组数，用B=2b表示所需要划分cache的内存块大小。

E：关联度（每组包含的缓存行数），输入指令后面的数据为关联度，即每一组有多少个块。

miss\_count：用于记录未命中的次数。

hit\_count：用于记录命中的次数。

eviction\_count：用于记录淘汰的次数。

### 1.3.3函数initCache()

功能：初始化cache。

实现方法：对于给定的参数S、B、E利用malloc函数动态开辟cache所需要的内存大小。实现代码如下：

void initCache(){

int i,j;

cache = (cache\_set\_t\*) malloc(sizeof(cache\_set\_t) \* S);

for (i=0; i<S; i++){

cache[i]=(cache\_line\_t\*) malloc(sizeof(cache\_line\_t) \* E);

for (j=0; j<E; j++){

cache[i][j].valid = 0;

cache[i][j].tag = 0;

cache[i][j].lru = 0;

}

}

set\_index\_mask = (mem\_addr\_t) (pow(2, s) - 1);

}

### 1.3.4函数freeCache()

功能：释放cache所占用的内存。

实现方法：用循环遍历cache所申请的指针数组内存块，以此free掉各个内存块。实现代码如下：

void freeCache(){

int i;

for(i=0;i<S;i++){

free(cache[i]);

}

free(cache);

}

### 1.3.5函数accessData()

功能：模拟cache进行的访问，给出所访问内存地址对应的是否命中或者是否淘汰。

实现方法：将输入地址和set\_index\_mask求与后得到一个索引，然后输入地址的前s+b位作为标签。依次查询每个块，并对比所需要的索引和标签，如果找到了相应数据，则hit\_count加1，否则miss\_count加1。并且在未命中时，先查询块中是否有空位置，有的话直接占用此空位置，否则需要找到lru最小的行（即最近访问时间间隔最长的行），然后进行数据的替换，eviction\_count加1。实现代码如下：

void accessData(mem\_addr\_t addr){

int i;

unsigned long long int eviction\_lru = ULONG\_MAX;

unsigned int eviction\_line = 0;

int haveEmpty=0;//是否含有空位置

mem\_addr\_t set\_index = (addr >> b) & set\_index\_mask;

mem\_addr\_t tag = addr >> (s+b);

int hit=0;

cache\_set\_t cache\_set = cache[set\_index];

for(i=0;i<E;i++)//E:相联度 即每一个组内有几块{

//所查询的块在cache内

if(cache\_set[i].tag==tag&&cache\_set[i].valid==1){

if(verbosity) printf("hit ");

hit\_count++;

cache\_set[i].lru = ULONG\_MAX;

hit = 1;

}

else if(!haveEmpty&&cache\_set[i].valid==0){

haveEmpty=1;

eviction\_line=i;

}

else if(cache\_set[i].valid==1){

//每进行一次查询，含有数据的块lru减1

cache\_set[i].lru--;

//找到lru最小的位置，作为预备淘汰

if(cache\_set[i].lru<eviction\_lru){

eviction\_lru=cache\_set[i].lru;

eviction\_line=i;

}

}

}

//未命中

if(hit==0){

if(verbosity) printf("miss ");

miss\_count++;

if(cache\_set[eviction\_line].valid==1)//有淘汰{

if(verbosity) printf("eviction ");

eviction\_count ++;

}

cache\_set[eviction\_line].valid=1;

cache\_set[eviction\_line].lru=ULONG\_MAX;

cache\_set[eviction\_line].tag=tag;

}

}

### 1.3.6函数replayTrace

功能：读取trace轨迹文件的内容，并根据其指令进行模拟内存访问的过程。

实现方法：通过文件读取函数fscanf进行文件流出读取，并将读取的内容赋值到operation、addr和len变量中，分别代表操作类型、地址、长度。然后根据operation的类型进行相应的函数调用。I指令忽略，S和L指令进行一次cache查询，即调用一次accessData函数，M指令进行两次cache查询。实现代码如下：

void replayTrace(char\* trace\_fn){

mem\_addr\_t addr=0;

unsigned int len=0;

char operation;

FILE\* trace\_fp = fopen(trace\_fn, "r");

while(fscanf(trace\_fp,"%c %llx,%d",&operation,&addr,&len)!=EOF){

if(operation=='I'&&verbosity)

printf("%c %llx,%u ",operation,addr,len);

else if(operation=='L'||operation=='S'||operation=='M'){

if(verbosity) printf("%c %llx,%u ",operation,addr,len);

accessData(addr);

if(operation=='M') accessData(addr);

if(verbosity) printf("\n");

}

}

fclose(trace\_fp);

}

### 1.3.7函数printUsage()

功能：当有输入格式错误或者输入-h指令时，输出相应的信息。

实现方法：用printf函数输出相关信息。实现代码如下：

void printUsage(char\* argv[]){

printf("Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n", argv[0]);

printf("Options:\n");

printf(" -h Print this help message.\n");

printf(" -v Optional verbose flag.\n");

printf(" -s <num> Number of set index bits.\n");

printf(" -E <num> Number of lines per set.\n");

printf(" -b <num> Number of block offset bits.\n");

printf(" -t <file> Trace file.\n");

printf("\nExamples:\n");

printf(" linux> %s -s 4 -E 1 -b 4 -t traces/yi.trace\n", argv[0]);

printf(" linux> %s -v -s 8 -E 2 -b 4 -t traces/yi.trace\n", argv[0]);

exit(0);

}

## 实验结果和分析

修改完成csim.c文件后，进行make编译，并运行test-csim，得到的测试结果如下：

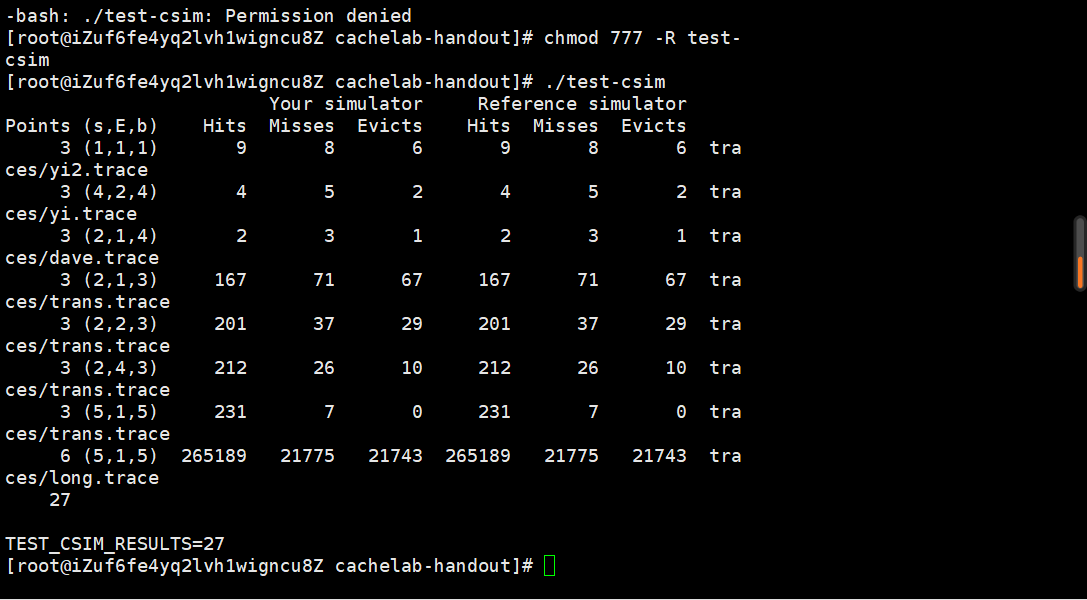


图 1-1 test-csim结果

由运行结果可知，cache模拟正确。

# 总结和体会

对cache进行的实验模拟算是比较简单的实验，仅仅只是模拟一个建议的cache然后对指令访问的命中和淘汰进行记录。需要考虑的函数实现大概就是accessData和replayTrace，前者要进行cache内数据的查找和替换，后者是对文件的读取和指令类型判断。总体上来说，完成cache模拟实验加深了对cache运行模式的认知，也很好的理解了LRU淘汰策略的工作机制。

# 对实验课程的建议

所给的实验说明比较少，初步读完后并不知道需要如何开始实验，建议多给一些相关的提示，如需要实现的函数构建、cache设计的基本思路等。

实验总体难度偏低，可以加大实验难度。