实验日志

1. 编写更新 Lru 数值的代码（10%）

```
//更新LruNumber，hit的话位最大打MAX_NUM,其他行的Lru均减1
void updataLruNumber(Sim_Cache *sim_cache, int setBits, int hitIndex)
{
        sim_cache ->sets[setBits].lines[hitIndex].LruNumber = MAX_NUM;
        for(int j = 0; j < sim_cache -> line_num; j++)
                for(j = 0; j < sim_cache -> line_num; j++)
                        if(j != hitIndex)
                                sim_cache -> sets[setBits].lines[j].LruNumber--;
}
```

如图：当某组中的某行 hit 或加载缓存成功时，将这一行的 Lru 计数值赋值为最大值，而该组中其他行 Lru 数值全部减 1。

2. 编写查找某组牺牲行函数的代码。（10%）

```
//查找某组中当前最小的LruNumber行，作为牺牲行
int findMinLruNumber(Sim_Cache *sim_cache, int setBits)
{
        int minIndex, minLru = MAX_NUM;
        for(int i = 0; i < sim_cache -> line_num; i++)
        {
                if(sim_cache -> sets[setBits].lines[i].LruNumber <= minLru)
                {
                        minIndex = i;
                        minLru = sim_cache -> sets[setBits].lines[i].LruNumber;
                }
        }
        return minIndex;
}
```

如图：遍历该组中的所有行，找出这组的 Lru 计数值最小的一行，作为牺牲行。

3. 编写命中判断的函数代码(10%）

```
//判断是否命中
int isMiss(Sim_Cache *sim_cache, int setBits, int tagBits)
{
        int Miss = 1;
        for(int i = 0; i < sim_cache -> line_num; i++)
        {
                if((sim_cache -> sets[setBits].lines[i].valid == 1)&&
                   (sim_cache -> sets[setBits].lines[i].tag == tagBits))
                {
                        Miss = 0;
                        updataLruNumber(sim_cache, setBits, i);
                }
        }
        return Miss;
}
```

如图：命中要符合：组索引匹配，标记位（tag）相同该行有效位为 1，否则就是不命中。命中后更新 Lru 计数值。

4. 编写更新高速缓存 cache 的函数代码（10%）

```
//更新高速缓存数据
int updateCache(Sim_Cache *sim_cache, int setBits, int tagBits)
{
        int i, full = 1;
        for(i = 0; i < sim_cache -> line_num; i++)
        {
                if(sim_cache -> sets[setBits].lines[i].valid == 0)
                {
                        full = 0;       break;
                }
        }
        if(full == 0)   //第i个有空位
        {
                sim_cache -> sets[setBits].lines[i].valid = 1;
                sim_cache -> sets[setBits].lines[i].tag = tagBits;
                updataLruNumber(sim_cache, setBits, i);
        }
        else                    //满了，驱逐
        {
                int evictionIndex = findMinLruNumber(sim_cache, setBits);
                sim_cache -> sets[setBits].lines[evictionIndex].valid = 1;
                sim_cache -> sets[setBits].lines[evictionIndex].tag = tagBits;
                updataLruNumber(sim_cache, setBits, evictionIndex);
        }
        return full;
}
```

如图：新加入的数据如果在组索引匹配位置有空位，就直接加入，否则找到 Lru 值最小的执行驱逐操作。组的每行的有效位全为 1 表示全满。加载新的数据块要更新有效位为 1，更新标记位，更新 Lru 计数值。

5. **检验 LRU 主函数代码编写与结果分析（20%）**

```
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 miss eviction
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:3 misses:6 evictions:4
```

```
setBits is 0, tagBits is 2
L 10,1 miss
setBits is 0, tagBits is 4
M 20,1 miss hit
setBits is 0, tagBits is 4
L 22,1 hit
setBits is 0, tagBits is 3
S 18,1 miss eviction
setBits is 0, tagBits is 22
L 110,1 miss eviction
setBits is 0, tagBits is 42
L 210,1 miss eviction
setBits is 0, tagBits is 2
M 12,1 miss eviction hit
hits:3 misses:6 evictions:4
```

如图:上图为正确的 csim-ref,下图为自己写的 csim，在 csim 中添加了一个组序列和标志位的输出语句，现在在结果正确的前提下可以通过输出信息进行 Lru 算法行为分析

L 10,1 中， setBits 为 0，tagBits 为 2 此时没有满，miss 后直接存入

M 20,1 中， setBits 为 0，tagBits 为 4 此时没有满，miss 后直接存入，第二次 hit

L 22,1 中， setBits 为 0，tagBits 为 4 此时 tag 与第二条相同，hit

S 18,1 中， setBits 为 0，tagBits 为 3 此时满员，miss 后驱逐第一条

L 110,1 中， setBits 为 0，tagBits 为 22 此时满员，miss 后驱逐第二条

L 210,1 中， setBits 为 0，tagBits 为 42 此时满员，miss 后驱逐第四条

M 12,1 中， setBits 为 0，tagBits 为 2 此时满员，miss 后驱逐第五条，第二次 hit

6. **编写加载数据的 L 命令处理函数代码（10%）**

```
//trace文件 L操作
void loadData(Sim_Cache *sim_cache, int setBits,int tagBits,int isVerbose)
{
        if(isMiss(sim_cache, setBits, tagBits) == 1)    //未命中
        {
                misses++;
                if(isVerbose == 1)
                        printf("miss ");
                if(updateCache(sim_cache, setBits, tagBits) == 1)//驱逐
                {
                        evictions++;
                        if(isVerbose == 1)
                                printf("eviction ");
                }
        }
        else                                    //命中
        {
                hits++;
                if(isVerbose == 1)
                        printf("hit ");
        }

}
```

如图：L 指令，本质是对数据的一次访问，首先在 cache 中检索是否命中，命中直接输出 hit，否则输出 miss，接着判断是否需要驱逐。

7. **编写存储数据的 S 命令处理函数和修改数据的 M 命令的处理函数（5%）**

```
//trace文件 S操作
void storeData(Sim_Cache *sim_cache, int setBits,int tagBits,int isVerbose)
{
        loadData(sim_cache, setBits, tagBits, isVerbose);
}
//trace文件 M操作
void modifyData(Sim_Cache *sim_cache,int setBits,int tagBits,int isVerbose)
{
        loadData(sim_cache, setBits, tagBits, isVerbose);
        storeData(sim_cache, setBits, tagBits, isVerbose);
}
```

S 也相当于对数据的一次访问，因此在 S 指令中调用一次 L 函数

而 M 指令相当于对数据的两次访问，因此在 M 指令中调用一次 L 函数和一次 S 函数

8. **编写获取 trace 脚本操作地址中的组索引与标记位的函数（10%）**

```
//获取地址中的组索引
int getSet(int addr, int s, int b)
{
        return (addr>>b)&((1<<s) - 1);
}
//获取地址中的标记位
int getTag(int addr, int s, int b)
{
        return addr>>(b+s);
}
```

函数如图所示，getSet 的原理是将数据右边 b 位的无关数据剔除，接着与 s 位长的”111…111”做与运算，保留了底 s 位。

getTag 的原理是将数据右边 b+s 位的无关数据剔除即可。

9. **检验 LRU 主函数代码编写与结果分析。（15%）**

```
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

```
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

如图：上图为 cism-ref 下图为 csim 在之前的 csim.c 基础上，进一步编写主函数，完全正确!