

# 计算机体系结构第四次实验

实验要求设计一个Cache模拟器，支持输入踪迹文件、cache大小、相联度、cache块大小，使用LRU算法处理替换发生时的情况，写失效时采用按写分配。

## 任务分析

输入的踪迹文件中包含三种访存的类型，行首的0、1、2分别代表读数据、写数据、读指令，每次读写数据时，数据的大小按照4B（32位）计算，每行的第二个分量是地址，按照32bit16进制数表示。读不命中需要将数据从更低一级的存储结构中调入cache，读不命中分为三种：Cold miss，Capacity miss，Conflict miss，三种方式发生的原因不同，但是所需要的处理方式是相同的；若发生了写不命中，会将数据从更低一级的存储结构中调入cache，在cache中完成写操作，这与读不命中的处理方式基本雷同，相异之处只有读写（以及失效）次数的统计；读指令可以看作是和读数据一样的处理方式，对于MIPS32架构的处理机，每次读指令的指令长度为32位，即4B。

## 逻辑设计

综上所述，我对于读指令、读数据、写数据三种访问cache的行为用了用一套逻辑来实现。下面是我的代码实现细节介绍：

- cache结构设计：

按照实验手册要求中的cache大小、相联度、块大小，可以计算出：如果采用预分配空间大小的方法，块的最大数量应该是 $64KB/16B = 4K = 4096$ 个，若采用直接相连，共有4096个组；如果使用8路组相连，一个组中最多有8行（即8个块），采用冗余度最高的方式分配空间如下：

- 每行cache表中包含两个整数，第一个整数存放cache的tag，第二个存放该行在整组中的“新旧度”，最旧访问的行的新旧度是1，随着访问越来越新依次递增；将这两个整数组为 `std::pair<int int>`，命名为LINE；
- 每个组中包含至多8行，使用 `std::vector<LINE>` 存储一个组的所有行的信息，命名为VLIN；
- 每个组还需要一个整数记录一下这个组中已被使用的行数，这个值只用来给每一行的新旧度变量赋值使用，能够简化时间复杂度，为了贴近真实，没有用这个整数来控制相联查找的复杂度，因为实际中的相联查找是同时查找了同一组中所有行；
- 将每个组与记录它已使用行数的整数构成一个 `std::pair<VLIN, int>`，命名为SET；
- 使用vector存储所有组，即： `std::vector<SET>`，命名为cache。

- cache读写逻辑设计：

依据上文分析，读指令、读写数据使用同一套操作逻辑，不同之处在于不同的访存操作需要记录在不同的统计信息里，操作逻辑是：

- 现将32位地址读入，依据块大小计算出块索引，再按照块索引和相联度计算出组索引与tag，在目标组处从头到尾遍历所有行，如果有一行的tag与计算出来的tag相同，则命中，否则不命中；
  - 命中时，需要修改该行的新旧度变量，设该行原本的新旧度是i，那么将该组中的其他新旧度大于i的行的新旧度减去1，将该行的新旧度设为组中已被使用行数。
- 如果不命中，需要遍历组中是否还有空闲空间，如果存在tag=-1（cache结构初始化时所有行的tag设为-1）的行，就将该行的tag标记为计算出来的tag，并标记其新旧度为最新的（即该组已使用行数），该操作意味着将数据从下一层级的存储器中拉入了cache；

- 如果没有找到空闲空间，就需要将组中最老的行替换出去，在这个设计中，最老的行的新旧度是1，而且保证任何一致的（即没有表项正在被替换或修改）时刻，每个组内如果有行被使用，那么一定包含一个新旧度是1的行；在替换时，和之前命中时类似，将所有新旧度大于1的行的新旧度减1，向原本新旧度为1的行中写入tag和新的新旧度。

- .din文件的处理：

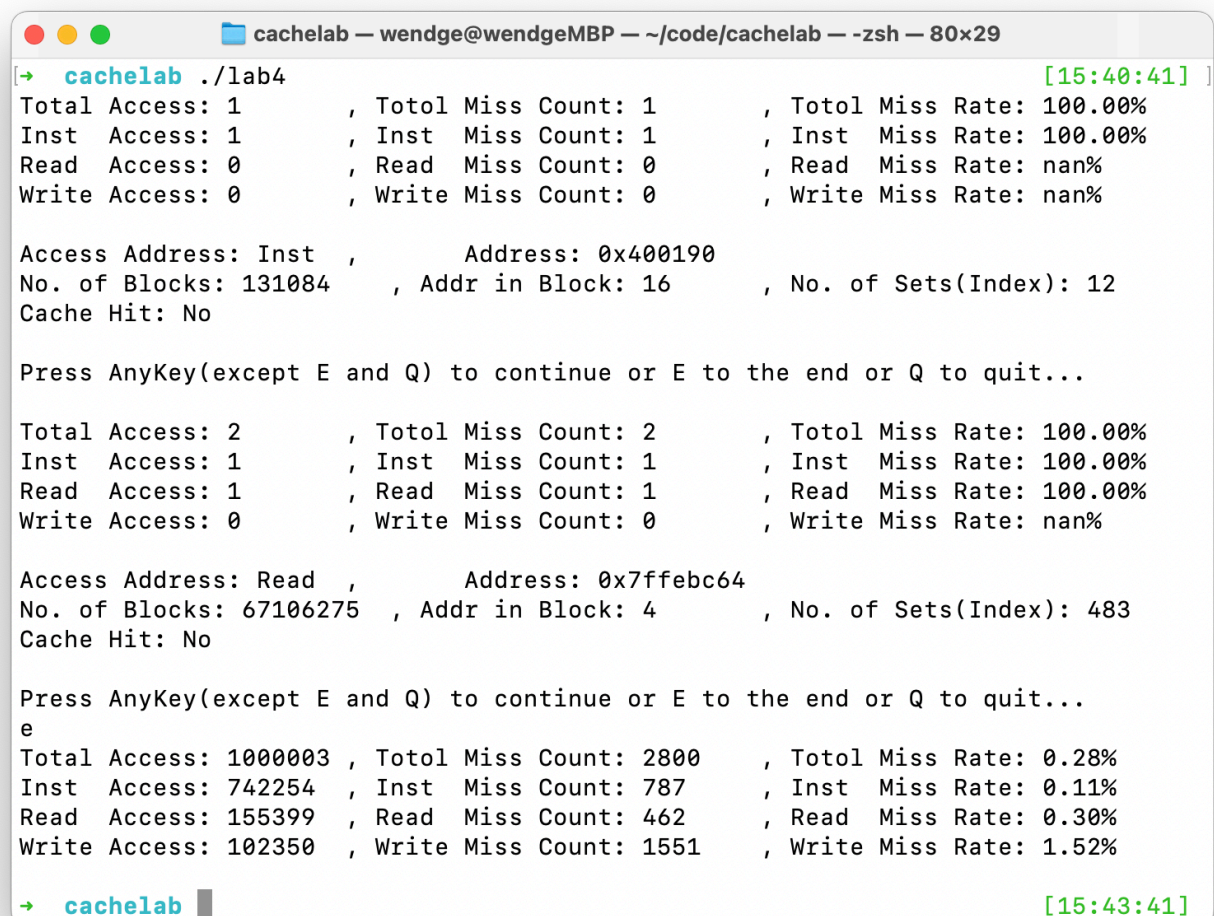
每次读入一行，将一行数据用空格分成多个token，将多个token放入一个vector中，返回给处理函数。

## 设计特色

- 使用g++编译器编译执行，可以使用 `--file_addr`、`--cache_size`、`--block_size`、`--cache_asso` 来指定相关参数，文件地址支持绝对地址或相对地址，不需要双引号引用；`cache_size`需要去掉末尾的K、KB等后缀，如要设置为32KB，写为32即可；`block_size`后面也不需要加B，要设置为16B的块大小，写16即可，`cache_asso`直接写入整数即可，下面是一个示例（假设可执行文件名称是 `lab4`）：

```
$ ./lab4 --file_addr tracefile/022.li.din --cache_size 64 --block_size 16 --cache_asso 2
```

- 使用两种控制方式控制程序前进：步进或执行到底，程序开始执行之后，若按下e或E键，即可执行到底，按下q或Q可以终止程序，按下其他任意键可以步进，执行界面也有相关提示：



```
→ cachelab ./lab4 [15:40:41]
Total Access: 1      , Totol Miss Count: 1      , Totol Miss Rate: 100.00%
Inst Access: 1      , Inst Miss Count: 1      , Inst Miss Rate: 100.00%
Read Access: 0      , Read Miss Count: 0      , Read Miss Rate: nan%
Write Access: 0     , Write Miss Count: 0     , Write Miss Rate: nan%

Access Address: Inst ,      Address: 0x400190
No. of Blocks: 131084 , Addr in Block: 16      , No. of Sets(Index): 12
Cache Hit: No

Press AnyKey(except E and Q) to continue or E to the end or Q to quit...

Total Access: 2      , Totol Miss Count: 2      , Totol Miss Rate: 100.00%
Inst Access: 1      , Inst Miss Count: 1      , Inst Miss Rate: 100.00%
Read Access: 1      , Read Miss Count: 1      , Read Miss Rate: 100.00%
Write Access: 0     , Write Miss Count: 0     , Write Miss Rate: nan%

Access Address: Read ,      Address: 0x7ffebc64
No. of Blocks: 67106275 , Addr in Block: 4      , No. of Sets(Index): 483
Cache Hit: No

Press AnyKey(except E and Q) to continue or E to the end or Q to quit...
e
Total Access: 1000003 , Totol Miss Count: 2800      , Totol Miss Rate: 0.28%
Inst Access: 742254  , Inst Miss Count: 787      , Inst Miss Rate: 0.11%
Read Access: 155399  , Read Miss Count: 462      , Read Miss Rate: 0.30%
Write Access: 102350  , Write Miss Count: 1551     , Write Miss Rate: 1.52%

→ cachelab [15:43:41]
```

# 源代码展示

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include <cstdint>

using namespace std;
typedef unsigned long long ull;
typedef pair<int, int> LINE;
typedef vector<LINE> VLINE;
typedef pair<VLINE, int> SET;
const int K = 1024;

// 默认参数
/*  cache_size: 8K, 16K, 32K, 64K;
   cache_asso: 1, 2, 4, 8;
   block_size: 16, 32, 64, 128 */
int cache_size = 64 * K; // 默认缓存大小
int cache_asso = 4; // 默认缓存关联度
int block_size = 32; // 默认块大小

int block_num = 0; // 块数
int set_num = 0; // 组数

ull read_cnt = 0;
ull write_cnt = 0;
ull inst_cnt = 0;
ull read_miss_cnt = 0;
ull write_miss_cnt = 0;
ull inst_miss_cnt = 0;

const int MAX_BLOCK_NUM = 64 * K / 16; // 最大块数 4096
const int MAX_SET_NUM = MAX_BLOCK_NUM / 1; // 最大组数 4096
const int MAX_LINE_NUM = 8; // 每组最大行数 8

// vector<pair<vector<pair<int, int>>, int>>
LINE tmp = make_pair(-1, 0);
vector<SET> cache(MAX_SET_NUM, SET(VLINE(MAX_LINE_NUM, tmp), 0));

vector<string> split(const string &s, char delimiter) {
    vector<string> tokens;
    string token;
    istringstream tokenStream(s);
    while (getline(tokenStream, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}
```

```

bool handler(uint32_t addr){
    int no_block = addr / block_size;
    int no_set = no_block % set_num;
    int tag = no_block / set_num;

    for(int i = 0; i < cache_asso; i++){
        if(cache[no_set].first[i].first == tag){
            int time = cache[no_set].first[i].second;
            for(int j = 0; j < cache_asso; j++){
                if(cache[no_set].first[j].second > time){
                    cache[no_set].first[j].second--;
                }
            }
            cache[no_set].first[i].second = cache[no_set].second;
            return true;
        }
    }
    // miss
    for(int i = 0; i < cache_asso; i++){
        if(cache[no_set].first[i].first == -1){
            // empty
            cache[no_set].first[i].first = tag;
            cache[no_set].first[i].second = ++(cache[no_set].second);
            return false;
        }
    }
    // full
    int min_index = 0;
    for(int i = 0; i < cache_asso; i++){
        if(cache[no_set].first[i].second == 1){
            min_index = i;
        }
        else{
            cache[no_set].first[i].second--;
        }
    }
    cache[no_set].first[min_index].first = tag;
    cache[no_set].first[min_index].second = cache[no_set].second;
    return false;
}

bool read_handler(uint32_t addr){
    read_cnt++;
    bool ret = handler(addr);
    if(!ret)
        read_miss_cnt++;
    return ret;
}

bool write_handler(uint32_t addr){
    write_cnt++;

```

```

    bool ret = handler(addr);
    if(!ret)
        write_miss_cnt++;
    return ret;
}

bool inst_handler(uint32_t addr){
    inst_cnt++;
    bool ret = handler(addr);
    if(!ret)
        inst_miss_cnt++;
    return ret;
}

void print(string op, uint32_t addr = 0, bool hit = false){
    printf("Total Access: %-8llu, Total Miss Count: %-8llu, Total Miss Rate: %3.2f%%\n",
        read_cnt + write_cnt + inst_cnt, read_miss_cnt + write_miss_cnt +
inst_miss_cnt,
        (double)(read_miss_cnt + write_miss_cnt + inst_miss_cnt) / (read_cnt +
write_cnt + inst_cnt) * 100.0);
    printf("Inst Access: %-8llu, Inst Miss Count: %-8llu, Inst Miss Rate: %3.2f%%\n",
        inst_cnt, inst_miss_cnt, (double)inst_miss_cnt / inst_cnt * 100.0);
    printf("Read Access: %-8llu, Read Miss Count: %-8llu, Read Miss Rate: %3.2f%%\n",
        read_cnt, read_miss_cnt, (double)read_miss_cnt / read_cnt * 100.0);
    printf("Write Access: %-8llu, Write Miss Count: %-8llu, Write Miss Rate: %3.2f%%\n",
        write_cnt, write_miss_cnt, (double)write_miss_cnt / write_cnt * 100.0);
    puts(" ");
    if(strcmp(op.c_str(), "end") == 0)
        return;
    if(strcmp(op.c_str(), "0") == 0)
        printf("Access Address: %-6s,          Address: 0x%x\n", "Read", addr);
    else if(strcmp(op.c_str(), "1") == 0)
        printf("Access Address: %-6s,          Address: 0x%x\n", "Write", addr);
    else
        printf("Access Address: %-6s,          Address: 0x%x\n", "Inst", addr);
    int no_block = addr / block_size;
    int no_set = no_block % set_num;
    int offset = addr % block_size;
    printf("No. of Blocks: %-10d, Addr in Block: %-8d, No. of Sets(Index): %-5d\n",
        no_block, offset, no_set);
    if(hit){
        printf("Cache Hit: %s\n", "Yes");
    }
    else{
        printf("Cache Hit: %s\n", "No");
    }
    puts(" ");
}

int main(int argc, char* argv[]) {
    string file_addr = "trace files/022.li.din"; // 默认文件地址
    // string file_addr = "trace files/test.din"; // 默认文件地址

```

```

for (int i = 1; i < argc; ++i) {
    string arg = argv[i];
    if (arg == "--file_addr" && i + 1 < argc) {
        file_addr = argv[++i];
    } else if (arg == "--cache_size" && i + 1 < argc) {
        cache_size = stoi(argv[++i]) * K;
    } else if (arg == "--block_size" && i + 1 < argc) {
        block_size = stoi(argv[++i]);
    } else if (arg == "--cache_asso" && i + 1 < argc) {
        cache_asso = stoi(argv[++i]);
    } else {
        cerr << "Unknown option: " << arg << endl;
    }
}

ifstream inputFile(file_addr);

if (!inputFile) {
    cerr << "Unable to open file " << file_addr;
}

block_num = cache_size / block_size; // 块数
set_num = block_num / cache_asso; // 组数

string line;
bool to_end = false;
while (getline(inputFile, line)) {
    vector<string> tokens = split(line, ' ');
    string op = tokens[0];
    uint32_t addr = stoul(tokens[1], 0, 16);
    bool hit = false;
    if (strcmp(op.c_str(), "0") == 0) { // Load Data (Read)
        hit = read_handler(addr);
    }
    else if (strcmp(op.c_str(), "1") == 0) { // Store Data (Write)
        hit = write_handler(addr);
    }
    else {
        hit = inst_handler(addr);
    }
    if (!to_end) {
        print(op, addr, hit);
        string next;
        cout << "Press AnyKey(except E and Q) to continue or E to the end or Q to quit..." << endl;
        getline(cin, next);
        if (strcmp(next.c_str(), "E") == 0 || strcmp(next.c_str(), "e") == 0) {
            to_end = true;
            continue;
        }
        else if (strcmp(next.c_str(), "Q") == 0 || strcmp(next.c_str(), "q") == 0) {
            break;
        }
    }
}

```

```

        }
    }
}
if(to_end){
    print("end");
}

inputFile.close();
return 0;
}

```

编译与运行方法：

环境：g++ 11或以上版本，linux环境或macOS环境：

```

$ cd cachelab
$ g++ lab4.cpp -o lab4
$ ./lab4 --file_addr tracefile/022.li.din --cache_size 64 --block_size 16 --cache_asso 2

```

环境：g++ 11或以上版本，Windows环境：

```

PS cd cachelab
PS g++ lab4.cpp -o lab4.exe
PS ./lab4.exe --file_addr tracefile/022.li.din --cache_size 64 --block_size 16 --
cache_asso 2

```

老师若要运行代码，请确保文件目录结构如下（run.sh 为测试脚本，请忽略）：

```

.
├── lab4.cpp
├── run.sh
└── tracefiles
    ├── 022.li.din
    ├── 047.tomcatv.din
    ├── 078.swm256.din
    └── 085.gcc.din

2 directories, 6 files

```

## 实验结果

使用提供的四种din文件，依次采用不同的cache大小、相联度、块大小，统计总体失效率：

022.li.din：



MissRate(%)	8KB				16KB				32KB				64KB			
	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8
16B	4.44	2.56	1.85	1.72	2.55	1.27	0.92	0.88	1.92	0.74	0.66	0.65	0.91	0.59	0.51	0.5
32B	3.64	1.98	1.31	1.16	1.99	0.97	0.57	0.52	1.53	0.49	0.37	0.36	0.62	0.38	0.28	0.28
64B	3.39	1.81	1.16	0.99	1.78	0.96	0.46	0.36	1.38	0.37	0.22	0.21	0.5	0.26	0.16	0.16
128B	3.89	1.73	1.22	0.96	1.84	0.91	0.39	0.27	1.42	0.3	0.14	0.13	0.44	0.2	0.09	0.09

047.tomcatv.din :

MissRate(%)	8KB				16KB				32KB				64KB			
	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8
16B	4.79	4.61	3.46	3.41	4.11	3.38	3.37	3.37	3.75	3.36	3.36	3.36	3.58	3.36	3.36	3.36
32B	2.73	2.59	1.83	1.75	2.24	1.75	1.71	1.71	1.99	1.71	1.71	1.71	1.88	1.71	1.71	1.71
64B	1.84	1.65	1.03	0.97	1.35	0.93	0.89	0.89	1.13	0.88	0.88	0.88	1.03	0.88	0.88	0.88
128B	1.7	1.39	0.63	0.56	1.07	0.52	0.47	0.48	0.77	0.47	0.47	0.47	0.64	0.47	0.47	0.47

078.swm256.din :

MissRate(%)	8KB				16KB				32KB				64KB			
	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8
16B	0.46	0.32	0.29	0.27	0.34	0.27	0.27	0.26	0.29	0.26	0.26	0.26	0.26	0.26	0.26	0.26
32B	0.33	0.19	0.17	0.16	0.22	0.14	0.14	0.14	0.17	0.14	0.14	0.14	0.14	0.14	0.14	0.14
64B	0.27	0.12	0.12	0.12	0.17	0.08	0.08	0.07	0.1	0.07	0.07	0.07	0.07	0.07	0.07	0.07
128B	0.3	0.09	0.09	0.09	0.18	0.05	0.05	0.04	0.09	0.04	0.04	0.04	0.04	0.04	0.04	0.04

085.gcc.din :

MissRate(%)	8KB				16KB				32KB				64KB			
	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8	Direct	2	4	8
16B	11.42	9.63	8.28	8.14	7.21	4.91	3.32	2.79	4.14	2.26	1.51	1.15	2.69	1.44	0.94	0.87
32B	8.54	7.33	6.55	6.4	5.57	3.93	3.09	2.7	3.15	1.8	1.13	0.8	1.95	0.97	0.56	0.5
64B	7.2	6.04	5.64	5.39	4.9	3.59	3.15	2.87	2.7	1.74	1.05	0.71	1.64	0.73	0.4	0.31
128B	6.87	5.26	5	4.8	4.75	3.56	3.15	3.02	2.6	1.84	1.14	0.98	1.54	0.67	0.43	0.23

如上表所示，同一个踪迹文件中：

- 当Cache大小和Block大小不变时，随着相联度增加，失效率逐渐降低。这是由于对块的位置的约束在逐渐减小，极端地，直接相联的缓存中，每个块的位置只有一个，而在全相联中，每个块的位置可以取遍整个cache。在 047.tomcatv.din 中，在缓存容量较大的情况下，2路组相连与更多路组相连的失效率没有较大区别，说明对于相联度已经达到饱和。
  - Cache大小为N的直接相联的失效率约等于Cache大小为N/2的二路组相联的失效率，可以从 022.li.din 的16KB直接相联与8KB二路组相联数据中找到证明。
- 当Cache大小和相联度不变时，随着块大小增大，失效率一般先减小，后增大。这是由于块的大小增大，虽然能减少Cold miss的次数，但是又会增多Conflict miss的次数，因为块的数量变少了，相同的块可能会不断被调入调出。
  - 将相同相联度、相同cache大小下，失效率最小的块大小命名为最优块大小，那么最优块大小会随着cache容量的增大而增大，这一点可以在 022.li.din 的直接相联统计数据或 078.swm.din 的直接相联统计数据中找到证明。



- 这可以得到一个小小的推论：都采用直接相联时，若小容量Cache的失效率随Block的增大只减小，那么说明最优Block大小还没有被取到，那么对于大容量Cache，做同样的一组Block大小的测试，其失效率也一定随着Block的增大而减小。本文得到的统计数据都能够佐证这一推论。
- 当相联度与Block大小不变时，Cache大小的增大会使失效率减小。047.tomcatv.din在相联度与Block大小不变的情况下，失效率基本保持不变，说明失效率对于Cache大小已经达到饱和。

## 实验感想

---

收获：本文中，我通过编写模拟程序，更深刻地了解了Cache的各个参数对于失效率的具体影响，加深了对于LRU算法的理解。

不足之处：本文只讨论了失效率这一评价指标，实际的Cache还要考虑不命中开销、命中时间等参数。命中开销包含LRU实现的复杂度，本文中软件实现了LRU算法，复杂度较高，但是灵活性强，若使用硬件实现，速度会更快。不命中开销需要考虑组中是否已满，还有存储总线的速度等参数。又由这些基本参数，可以计算出最终的平均访存时间AMAT。实际上的Cache性能强弱分析应该是以平均访存时间为指标的。