Cache Lab Report

Part A: Writing a Cache Simulator

这个部分中，我们需要写一个程序来模拟缓存对于给定内存访问序列的工作情况。

首先定义缓存行如下：

```
struct Line {
    int valid;
    long long tag;
    int timestamp;
};
```

valid和tag意义与教材中一致（地址有64位，tag有可能超过32位的int范围）。timestamp是最近访问这个行的时间，用于在替换时进行LRU替换。由于只是模拟，没有必要真的用一个`char buf[B]`记录缓存内容，比对时可以忽略word selection这一步，仅需确认访问的数据是否在行内即可。

缓存下数据的访问模拟如下：

```
void access_data(long long addr)
{
    int i, j;
    long long tag;
    // set selection
    i = (addr >> b) & ((1 << s) - 1);
    // line matching
    tag = (addr >> (b+s)) & ((1ll << t) - 1);
    for (j = 0; j < E; j++) {
        if (!cache[i][j].valid) continue;
        if (cache[i][j].tag != tag) continue;
        cache[i][j].timestamp = time++;
        hit++;
        return;
    }
    miss++;
    emplace_data(i, tag);
}
```

计算出set的index，再到对应set中进行行比对，寻找valid并且tag match的行。如果找到了就是一次hit，更新时间戳；否则说明数据不在内存里，记一次miss，这时需要把数据放入内存：

```
void emplace_data(int i, long long tag)
{
    int j, tmn;
    for (j = 0; j < E; j++) {
        if (cache[i][j].valid) continue;
        cache[i][j].valid = 1;
        cache[i][j].tag = tag;
```

```
            cache[i][j].timestamp = time++;
            return;
        }
        // LRU replacement
        tmn = time;
        for (j = 0; j < E; j++) {
            if (cache[i][j].timestamp < tmn)
                tmn = cache[i][j].timestamp;
        }
        for (j = 0; j < E; j++) {
            if (cache[i][j].timestamp != tmn) continue;
            cache[i][j].tag = tag;
            cache[i][j].timestamp = time++;
        }
        eviction++;
    }
```

首先寻找是否有空行，如果有就可以直接把数据放进去；否则说明这个set已经满了，需要evict一行，使用LRU策略意味着需要找到上次访问最早的一行，此时遍历找到时间戳最小的行进行替换，同时记一次eviction。

主函数中主要涉及参数的parsing，可以借助`getopt`实现：

```
while ((opt = getopt(argc, argv, "s:E:b:t:")) != -1) {
    switch (opt) {
    case 's':
        s = atoi(optarg);
        break;
    case 'E':
        E = atoi(optarg);
        break;
    ...
    }
}
```

`argc`和`argv`是运行程序时系统传给主程序的参数，分别是参数数量和参数数组，按照惯例第一个参数即`argv[0]`为程序名。

还有valgrind记录的读取（同时在线模拟），通过`fgets`和`sscanf`实现：

```
while (fgets(buf, 19, in) != NULL) {
    if (buf[0] == 'I') continue;
    sscanf(buf, " %c %llx,%d", &type, &addr, &size);
    if (type == 'M')
        access_data(addr);
    access_data(addr);
}
```

M（Modify）对应读写两次内存访问，其他的L（Load）和S（Store）涉及一次。

Part B: Optimizing Matrix Transpose

这个部分要求我们对三个特定大小的矩阵转置进行优化，减小cache miss次数。

缓存参数为 $(s = 5, E = 1, b = 5)$ ，说明这是一个有32个set，每个set有一个32个byte的block的directly-mapped缓存。每个block可以容纳8个int矩阵元素。

32 x 32

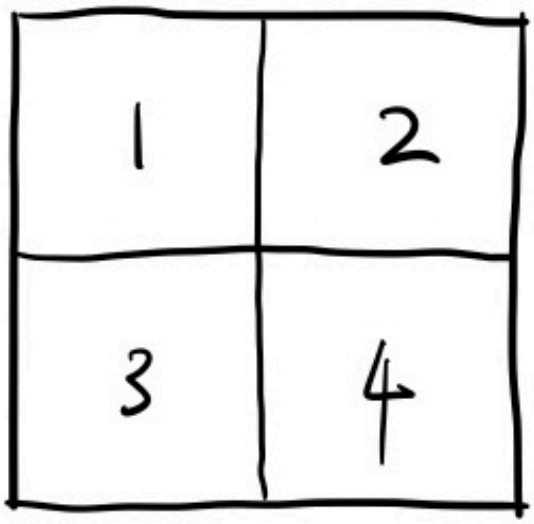
由于矩阵尺寸正好是32，A和B一些对应位置的元素会被映射到同一个set里造成conflict miss（写时evict读时fetch的block），所以每次读一个元素就要尽可能把这个block的元素读完。题目允许我们使用至多12个本地变量，因此除了用来遍历的索引变量可以用8个变量作为补充缓存，把读到的元素存起来（本地变量一般会被放在寄存器中，读写cycle数少于缓存）。

要实现这一流程，我们可以把矩阵分成8 * 8的分块来处理，遍历一个分块的所有行，每次把一行的8个元素存入本地变量，全部读完再写入对应的位置。

64 x 64

此时矩阵一行占8个block，每4行就会填满cache，相差4行的元素即可产生冲突。如果继续使用8 * 8的分块，写入B时每列下面四个就会和上面的冲突；使用4 * 4的分块又会导致每个放入缓存的block中有1/2的内容没有得到利用。所以我们需要使用一点技巧。

把8 * 8的块分成四个4 * 4的，按行优先编为1234，如图所示。



考虑A和$B = A^T$中的对应块：A中的1和4转置后会到B中对应的1和4，而2和3则是分别到3和2。题目要求A只读B可修改，因此先把A中的1和2分别转移到B中的1和2（没有冲突；B中2上下翻转）。接下来同时按列读3（从右往左）和4（从左往右），在这一过程中从下往上把B中刚才移过来的2移到正确的位置，每轮迭代移好的2和4正好在同一行中。

61 x 67

此时矩阵宽度为67，不存在特定位置对应元素的冲突，可以直接用16 * 16的分块，特判一下仍有冲突的对角线位置的元素。