实验报告

Part A

首先观察发现模拟cache包括几个方面，读取cache参数，通过参数初始化cache，读取并执行指令集，析构cache，最后打印结果。所以可以将程序分成几个函数来实现。

getParameters

实现思路

通过实验指导可以知道，cache参数是通过命令行参数进行传递的时候通过库函数getopt得到每次的指令，之后通过switch为cache设置参数。

C代码

```
void getParameters(int argc, char** argv) {
    int option;
    while((option = getopt(argc, argv, "h:v:s:E:b:t:u")) != -1) {
        switch (option) {
            case 'h':
                printHelpInformation();
                exit(0);
            case 'v':
                isTrace = true;
```

```
                break;
            case 's':
                s = atoi(optarg);
                S = 1 << s;
                break;
            case 'E':
                E = atoi(optarg);
                break;
            case 'b':
                b = atoi(optarg);
                B = 1 << b;
                break;
            case 't':
                fp = fopen(optarg, "r");
                strcpy(fileName, optarg);
                break;
            default:
                printf("Error option!\n");
                break;
        }
    }
}
```

initializeCache

实现思路

首先通过设置的参数为cache分配空间，之后对于每一个set中的每一个line，设置其为从未使用过的状态，即tag为0，vaild为不可用，timestamp为-1。最后将指针指向分配的空间。

C代码

```
void initializeCache() {
    cache = malloc(S * sizeof(void*));
    for (int i = 0; i < S; i ++ ) {
        cache[i] = malloc(sizeof(void*) * E);
        for (int j = 0; j < E; j ++ ) {
            Line line = malloc(sizeof(Block));
            line->tag = 0;
            line->valid = false;
            line->last = -1;
            cache[i][j] = line;
        }
    }
}
```

getOperations

实现思路

首先需要从文件中读取指令，之后通过指令的类型来判断需要调用几次cache。load与store指令需要调用一次cache，而modify指令需要调用两次cache。

C代码

```
void getOperations() {
    while(fgets(option, N, fp)) {
        timeStamp ++ ;
        Operation operation;
        int count = sscanf(option, " %c%llx,%d", &operation.type,
&operation.address, &operation.bias);
        if (operation.type == 'I' || !count) continue;
        if (isTrace) puts("option");

        if (operation.type == 'M') tryHit(operation);
        tryHit(operation);
    }
}
```

tryHit

实现思路

首先取出指令中的地址用于访问cache，通过指令中的地址可以得到内存中的实际地址与访问cache中的set偏移量，之后检查set中的每一个line，如果有line中的tag与访问内存中的地址相同的则代表命中，否则代表不命中。之后检查是否有空的line，将空的line的tag设为目标地址，否则通过LRU规则将timestamp最小的line进行替换，并将替换数增加。

C代码

```
void tryHit(Operation operation) {
    Address address = operation.address;
    int setBias = (address >> b) & ((1 << s) - 1);
    int tagAddress = address >> (s + b);

    Set set = cache[setBias];
    Line hitBlock = NULL;
    Line emptyBlock = NULL;
    Line LRUBlock = NULL;

    for (int i = 0; i < E; i ++ ) {
        Line line = set[i];
        if (!hitBlock && line->valid && line->tag == tagAddress) hitBlock =
line;
        if (!emptyBlock && !line->valid) emptyBlock = line;
        if (line->valid && (!LRUBlock || LRUBlock->last > line->last)) LRUBlock
= line;
    }

    if (hitBlock) {
        hitBlock->last = timeStamp;
        if (isTrace) puts("hit!");
        hitCount ++ ;
    } else {
        missCount ++ ;
        if (isTrace) puts("miss!");
        if (emptyBlock) {
```

```
            emptyBlock->last = timeStamp;
            emptyBlock->valid = true;
            emptyBlock->tag = tagAddress;
        } else {
            evictionCount ++ ;
            if (isTrace) puts("eviction!");
            LRUBlock->last = timeStamp;
            LRUBlock->valid = true;
            LRUBlock->tag = tagAddress;
        }
    }
}
```

Part B

首先观察得知该实验中用的模拟cache的参数为S = 32， E = 1， B = 32,总容量为1KB。之后通过不同矩阵的参数尺寸设计hit率最高的举证转置算法。

32*32

分析

对于矩阵A而言，它是按行遍历的，空间局部性较好，一共会造成$32 * 8 = 128$次miss，而对于矩阵B，它是按列遍历的，而对于$32 * 32$的矩阵，通过计算得知，cache中可以最多容纳8行的数据，所以每一次都会造成一次miss，共会造成1024次miss。所以提高hit率的关键是降低矩阵B的miss。因为cache中最多容纳8行数据，所以可以将矩阵分成16个$8 * 8$的矩阵。理论上共会造成$2 * 8 * 16 = 256$次miss。而实际的miss数大于理论值，是因为矩阵A与矩阵B是共用一个cache，并且通过观察发现，他们起始地址映射到cache中的地址相同。所以当分块在对角线上的时候，会因为A与B的行之间相互冲突而造成双倍的miss。对于这种情况，这里可以提高时间局部性，利用寄存器存储A中一行的值，这样A中这一行的元素就不需要再次访问了，可以有效地较少miss数。

C代码

```
for (int i = 0; i < 32; i += 8 ) // 8*8的分块
    for (int j = 0; j < 32; j += 8 )
        for (int k = i; k < i + 8; k ++ ) { // 局部变量提高时间局部性
            int t0 = A[k][j + 0], t1 = A[k][j + 1];
            int t2 = A[k][j + 2], t3 = A[k][j + 3];
            int t4 = A[k][j + 4], t5 = A[k][j + 5];
            int t6 = A[k][j + 6], t7 = A[k][j + 7];
            B[j + 0][k] = t0;
            B[j + 1][k] = t1;
            B[j + 2][k] = t2;
            B[j + 3][k] = t3;
            B[j + 4][k] = t4;
            B[j + 5][k] = t5;
            B[j + 6][k] = t6;
            B[j + 7][k] = t7;
        }
```

64*64

分析

对于$64*64$的矩阵而言，cache中每次最多可以存下4行的数据，若采取与之前一样分块的方法需要分成$4*4$的矩阵，一共会造成$4*256+4*128=1536$次miss。之所以造成这种情况是因为之前cache中缓存了8个字节的数据只用到了4个，造成了cache资源的浪费。针对这种情况，我们首先将分块的基本单元定为$8*8$的矩阵。对于每一个$8*8$的矩阵，通过上面的方法转置左上角$4*4$的矩阵，之后将右上角$4*4$的矩阵转置完放到B的右上角。对于A左下角的$4*4$矩阵，每一行分别处理。首先将之前B右上角暂存的结果挪到左下角，之后再将A左下角的$4*4$的矩阵转置放回B右上角，之后再将A的右下角的$4*4$矩阵转置放置到B的右下角。

C代码

```
for (int i = 0; i < 64; i += 8 )
            for (int j = 0; j < 64; j += 8 ) {
                for (int k = 0; k < 4; k ++ ) { // step 1, 2
                    int t0 = A[i + k][j + 0], t1 = A[i + k][j + 1];
                    int t2 = A[i + k][j + 2], t3 = A[i + k][j + 3];
                    int t4 = A[i + k][j + 4], t5 = A[i + k][j + 5];
                    int t6 = A[i + k][j + 6], t7 = A[i + k][j + 7];
                    B[j + 0][i + k] = t0;
                    B[j + 1][i + k] = t1;
                    B[j + 2][i + k] = t2;
                    B[j + 3][i + k] = t3;
                    B[j + 0][i + 4 + k] = t4;
                    B[j + 1][i + 4 + k] = t5;
                    B[j + 2][i + 4 + k] = t6;
                    B[j + 3][i + 4 + k] = t7;
                }

                for (int k = 0; k < 4; k ++ ) { // step 3
                    int t0 = A[i + 4][j + k], t1 = A[i + 5][j + k];
                    int t2 = A[i + 6][j + k], t3 = A[i + 7][j + k];
                    int t4 = B[j + k][i + 4], t5 = B[j + k][i + 5];
                    int t6 = B[j + k][i + 6], t7 = B[j + k][i + 7];
                    B[j + k][i + 4] = t0;
                    B[j + k][i + 5] = t1;
                    B[j + k][i + 6] = t2;
                    B[j + k][i + 7] = t3;
                    B[j + k + 4][i + 0] = t4;
                    B[j + k + 4][i + 1] = t5;
                    B[j + k + 4][i + 2] = t6;
                    B[j + k + 4][i + 3] = t7;
                }

                for (int k = 0; k < 4; k ++ ) { // step 4
                    int t0 = A[i + k + 4][j + 4], t1 = A[i + k + 4][j + 5];
                    int t2 = A[i + k + 4][j + 6], t3 = A[i + k + 4][j + 7];
                    B[j + 4][i + k + 4] = t0;
                    B[j + 5][i + k + 4] = t1;
                    B[j + 6][i + k + 4] = t2;
                    B[j + 7][i + k + 4] = t3;
                }
```

```
            }
```

61*67

分析

对于这种不规则的矩阵，无法准确分出每一块，只能通过不断地对分块的步长进行试探，从len=2开始，到len=20，得出当N为17的时候miss总数最小。

C代码

```
int len = 17;
for (int i = 0; i < N; i += len)
    for (int j = 0; j < M; j += len)
        for (int k = i; k < N && k < i + len; k ++ )
            for (int h = j; h < M && h < j + len; h ++ )
                B[h][k] = A[k][h];
```