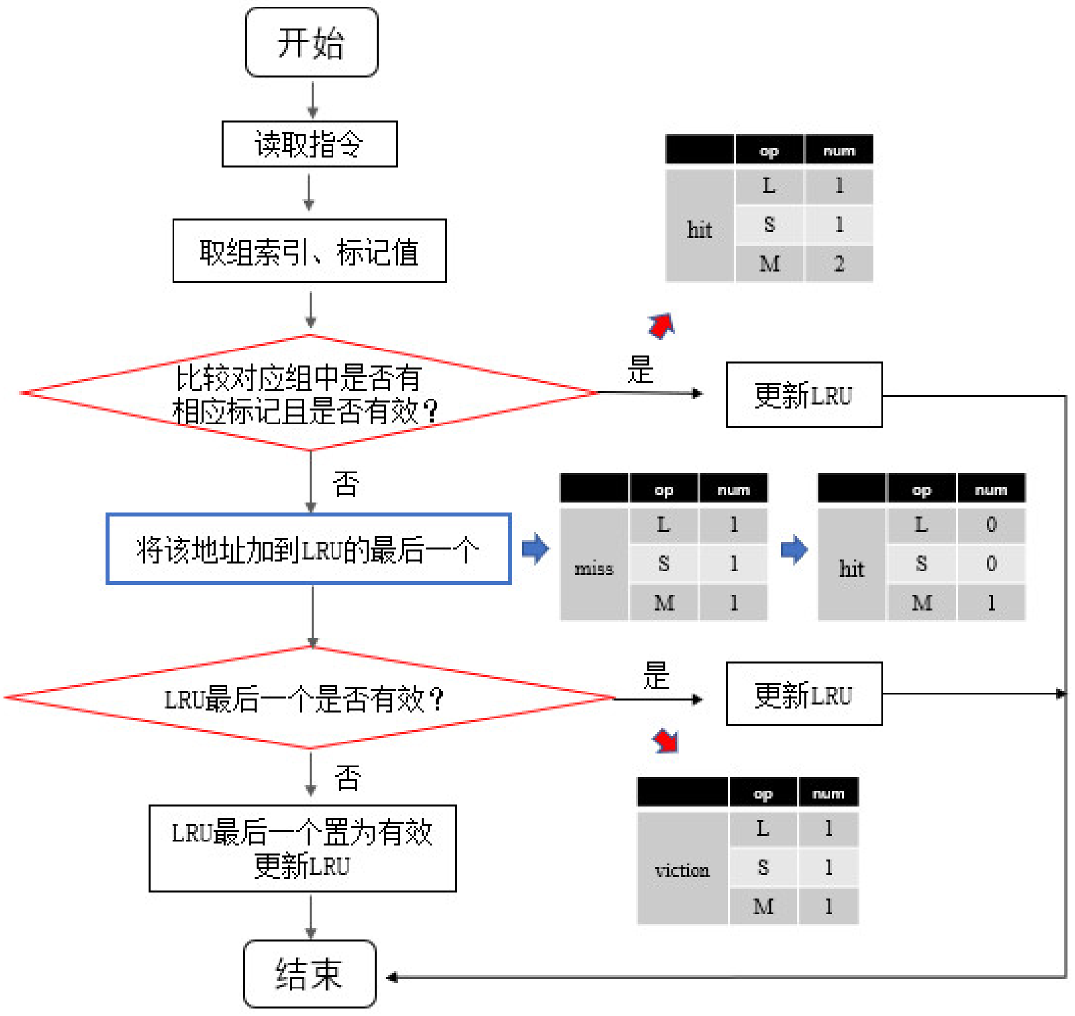
CSAPPLab5 CacheLab

通过本实验可以帮助我们理解缓存对于 *c* 程序性能的影响。本次实验包含两个部分，第一部分是写一段 *c* 程序（大约 *200-300* 行）模拟缓存的运行机制；第二部分是优化一个小的矩阵转置函数，以最小化缓存不命中的次数。

Part A: Writing a Cache Simulator

该部分实验要求编写高速缓存模拟函数 *csim.c*，以给定的内存追踪文件作为输入，模拟缓存的 *hit/miss* 的行为，输出最终 *hit, miss* 和 *eviction* 的总次数，并要求采用最近最少被使用（*LRU*）替换策略。程序结构图如下：



- 读取指令操作即为读取输入的内存追踪文件，文件内容格式如下：

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

注意：*"I"* 前面无空格，*"M", "L", "S"* 前有空格。

相关代码如下（注：因读取文件行数不定，采用动态分配内存空间不足时需重新申请更大的空间）：

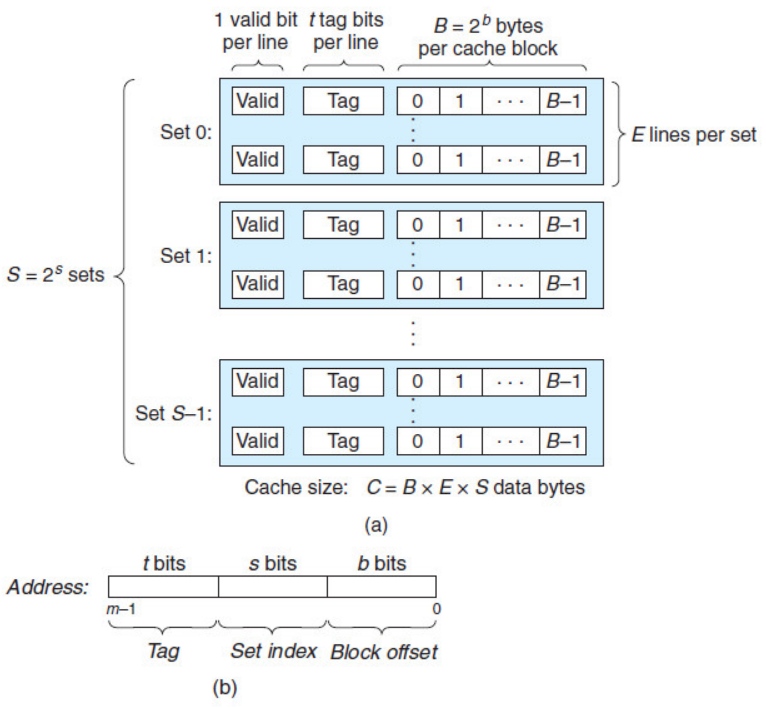
```c
const int NUM = 16;

typedef struct {
    char identifier;
    unsigned address;
    int size;
}Instruct;

typedef struct {
    Instruct* instruct_p;
    int length;
}InstructPoint;

InstructPoint malloc_instruct(InstructPoint instruct_point, int length)
{
    InstructPoint p;
    p.length = length << 2;
    p.instruct_p = (Instruct *)malloc(p.length * sizeof(Instruct));
    memcpy(p.instruct_p, instruct_point.instruct_p, length *
sizeof(Instruct));
    free(instruct_point.instruct_p);
    instruct_point.instruct_p = NULL;
    return p;
}

InstructPoint readInstruct(const char* filename) {
    int count = 0;

    InstructPoint instruct_point;
    instruct_point.length = NUM;
    instruct_point.instruct_p = (Instruct
*)malloc(instruct_point.length * sizeof(Instruct));

    FILE *pFile;    // pointer to FILE object

    pFile = fopen(filename, "r");   // open file for reading

    char identifier;
    unsigned address;
    int size;

    // Reading lines like " M 20£¬1" or "L 19,3"
    while (fscanf(pFile, " %c %x, %d", &identifier, &address, &size) >
0) {
        instruct_point.instruct_p[count].identifier = identifier;
        instruct_point.instruct_p[count].address = address;
        instruct_point.instruct_p[count].size = size;
        count++;
        if (count >= instruct_point.length) {
            instruct_point = malloc_instruct(instruct_point,
instruct_point.length);
        }
        //printf("%c %x, %d\n", identifier, address, size);
    }
    instruct_point.length = count;

    fclose(pFile);

    return instruct_point;
}
```

- 初始化每条指令三种行为 *hit, miss, eviction* 数目均为 *0*， 代码如下：

```
typedef struct {
    short hit;
    short miss;
    short eviction;
}State;

typedef struct {
    State* state_p;
    int length;
}StatePoint;

StatePoint initState(int size) {
    StatePoint state_point;
    state_point.length = size;
    state_point.state_p = (State *)malloc(state_point.length * 
sizeof(State));
    for (int i = 0; i < state_point.length; i++) {
        state_point.state_p[i].hit = 0;
        state_point.state_p[i].miss = 0;
        state_point.state_p[i].eviction = 0;
    }
    return state_point;
}
```

- 高速缓存 *(S, E, B, m)* 的通用组织结构如下图所示，相关内容见书籍《深入理解计算机系统》第426页。

Figure 6.25
General organization of cache (S, E, B, m).
(a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.



程序的输入参数包含：组索引位个数 *s* ， 每个组包含高速缓存行的个数 *E* 和数据块的位个数 *b* ，三者大小均可任意输入。易知高速缓存组的个数 *S = 1 << s* ， 每个数据块中包含字节的个数 *B = 1 << b* 。

根据上图，给定地址 *address* ， 求出标记位（*Tag*）， 组索引（*Set index*）和块偏移（*Block offset*）的函数分别为：

```c
unsigned getTag(unsigned address, int b, int s) {
    return (address >> (b + s));
}

unsigned getSet(unsigned address, int b, int s) {
    unsigned mask_set = (1 << s) - 1;
    return ((address >> b) & mask_set);
}

unsigned getBlock(unsigned address, int b) {
    unsigned mask_block = (1 << b) - 1;
    return (address & mask_block);
}
```

- 以二维数组 *cache[S][E]* 代表缓存，每个元素包含有效位和标记位，均初始化为 *0* 。代码如下：

```c
typedef struct {
    int valid;
    unsigned tag;
}Cache;

Cache** initCache(int S, int E) {
    Cache** cache = (Cache **)malloc(S * sizeof(Cache*));
    for (int i = 0; i < S; i++) {
        cache[i] = (Cache *)malloc(E * sizeof(Cache));
    }
    for (int i = 0; i < S; i++) {
        for (int j = 0; j < E; j++) {
            cache[i][j].valid = 0;
            cache[i][j].tag = 0;
        }
    }
    return cache;
}
```

- 缓存替换策略 *LRU* 采用数组 *LRU[S][E]* 表示，数组行数代表高速缓存组数 *S* ， 数组列数代表每个组包含的高速缓存行数 *E* ， 数组每行元素分别存储该高速缓存组包含的高速缓存行数的索引，即 *0, 1, 2, ... , E-1* ， 每行元素最后一个值存储的即为该高速缓存组最近最少使用的的高速缓存行的索引。所以，每次进行替换时，只需与相应高速缓存组最后一个元素值对应的高速缓存行进行替换即可（对数组 *cache[S][E]* 操作）， 并更新数组 *LRU[S][E]* 将其该行的最后一个元素值放至首位，并对其它元素依次后移（注：即使不进行替换，即缓存命中 *hit* 的情况，也许按该种规则更新数组 *LRU[S][E]* 。相关代码如下：

```c
int** initLRU(int S, int E) {
    int** LRU = (int **)malloc(S * sizeof(int*));
    for (int i = 0; i < S; i++) {
        LRU[i] = (int *)malloc(E * sizeof(int));
    }
    for (int i = 0; i < S; i++) {
        for (int j = 0; j < E; j++) {
                LRU[i][j] = j;
            }
        }
    return LRU;
}

void reorderLRU(int** LRU, unsigned set, int val) {
    int id;
    for (id = 0; ;id++){
        if (LRU[set][id] == val){
            break;
        }
    }
    int tmp = LRU[set][id];
    for (int i = id; i > 0; i--) {
        LRU[set][i] = LRU[set][i - 1];
    }
    LRU[set][0] = tmp;
}
```

- 按照结构图中流程，判断各指令行为的核心代码如下：

```
void evalState(InstructPoint instruct_point, Cache** cache_p, int**
LRU, int s, int E, int b, StatePoint state_point) {
    char identifier;
    unsigned address;
    //int size;

    unsigned set;
    unsigned tag;

    int k;
    for (int i = 0; i < instruct_point.length; i++) {
        identifier = instruct_point.instruct_p[i].identifier;
        address = instruct_point.instruct_p[i].address;
        //size = instruct_point.instruct_p[i].size;
        switch (identifier) {
        case 'I':
            break;
        case 'L':
        case 'S':
        case 'M':
            set = getSet(address, b, s);
            tag = getTag(address, b, s);
            for (k = 0; k < E; k++) {
                if (cache_p[set][k].valid && (cache_p[set][k].tag ==
tag)) {//命中
                    state_point.state_p[i].hit = 1;
                    if (identifier == 'M') {
                        state_point.state_p[i].hit = 2;
                    }
                    reorderLRU(LRU, set, k);
                    break;
                }
            }
            if (k == E) { // 不命中
                state_point.state_p[i].miss = 1;
```

```
                    if (identifier == 'M') {
                        state_point.state_p[i].hit = 1;
                    }
                    cache_p[set][LRU[set][E - 1]].tag = tag;
                    if (cache_p[set][LRU[set][E - 1]].valid) {
                        state_point.state_p[i].eviction = 1;
                    }
                    else {
                        cache_p[set][LRU[set][E - 1]].valid = 1;
                    }
                    reorderLRU(LRU, set,LRU[set][E - 1]);
                }
                break;
            default:
                printf("Illegal Instruction!!");
                break;
            }
        }
}
```

附：完整代码

```
#include "cachelab.h"
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>

const int NUM = 16;

typedef struct {
    char identifier;
    unsigned address;
    int size;
}Instruct;

typedef struct {
    Instruct* instruct_p;
    int length;
}InstructPoint;

typedef struct {
    short hit;
    short miss;
    short eviction;
}State;

typedef struct {
    State* state_p;
    int length;
}StatePoint;

typedef struct {
    int valid;
    unsigned tag;
```

```
}Cache;

InstructPoint malloc_instruct(InstructPoint instruct_point, int length);
InstructPoint readInstruct(const char* filename);
StatePoint initState(int size);
Cache** initCache(int S, int E);
int** initLRU(int S, int E);
unsigned getBlock(unsigned address, int b);
unsigned getSet(unsigned address, int b, int s);
unsigned getTag(unsigned address, int b, int s);
void reorderLRU(int** LRU, unsigned set, int id);
//void printSummary(int hit_count, int miss_count, int eviction_count);
void evalState(InstructPoint instruct_point, Cache** cache_p, int** LRU, int s,
int E, int b, StatePoint state_point);

int main(int argc, char** argv)
{
    int opt, s, E, b;
    char* t = NULL;
    while(-1 != (opt = getopt(argc, argv, "hvs:E:b:t:"))){
        switch (opt){
            case 's':
            s = atoi(optarg);
            break;
            case 'E':
            E = atoi(optarg);
            break;
            case 'b':
            b = atoi(optarg);
            break;
            case 't':
            t = optarg;
            break;
            default:
            printf("wrong argument\n");
            break;
        }
    }
    int S = 1 << s;
    //int B = 1 << b;

    // instruct initiation
    const char* filename = t;
    InstructPoint instruct_point;
    instruct_point = readInstruct(filename);

    // state initiation
    StatePoint state_point;
    state_point = initState(instruct_point.length);

    // cache initiation
    Cache** cache_p = NULL;
    cache_p = initCache(S, E);

    // LRU initiation
    int** LRU = NULL;
    LRU = initLRU(S, E);
```

```c
    // process
    evalState(instruct_point, cache_p, LRU, s, E, b, state_point);

    // print
    int hit_count = 0;
    int miss_count = 0;
    int eviction_count = 0;
    for (int i = 0; i < state_point.length; i++) {
        hit_count += state_point.state_p[i].hit;
        miss_count += state_point.state_p[i].miss;
        eviction_count += state_point.state_p[i].eviction;
        //printf("%d: hit = %d, miss = %d, eviction = %d\n", i,
state_point.state_p[i].hit, state_point.state_p[i].miss,
state_point.state_p[i].eviction);
    }

    // free
    free(instruct_point.instruct_p);
    free(state_point.state_p);
    for (int i = 0; i < S; i++) {
        free(cache_p[i]);
        free(LRU[i]);
    }
    free(cache_p);
    free(LRU);

    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}

InstructPoint malloc_instruct(InstructPoint instruct_point, int length) {
    InstructPoint p;
    p.length = length << 2;
    p.instruct_p = (Instruct *)malloc(p.length * sizeof(Instruct));
    memcpy(p.instruct_p, instruct_point.instruct_p, length * sizeof(Instruct));
    free(instruct_point.instruct_p);
    instruct_point.instruct_p = NULL;
    return p;
}

InstructPoint readInstruct(const char* filename) {
    int count = 0;

    InstructPoint instruct_point;
    instruct_point.length = NUM;
    instruct_point.instruct_p = (Instruct *)malloc(instruct_point.length *
sizeof(Instruct));

    FILE *pFile;    // pointer to FILE object

    pFile = fopen(filename, "r");   // open file for reading

    char identifier;
    unsigned address;
    int size;

    // Reading lines like " M 20£¬1" or "L 19,3"
    while (fscanf(pFile, " %c %x, %d", &identifier, &address, &size) > 0) {
        instruct_point.instruct_p[count].identifier = identifier;
        instruct_point.instruct_p[count].address = address;
        instruct_point.instruct_p[count].size = size;
        count++;
        if (count >= instruct_point.length) {
            instruct_point = malloc_instruct(instruct_point,
instruct_point.length);
        }
        //printf("%c %x, %d\n", identifier, address, size);
    }
    instruct_point.length = count;

    fclose(pFile);

    return instruct_point;
}

StatePoint initState(int size) {
    StatePoint state_point;
    state_point.length = size;
    state_point.state_p = (State *)malloc(state_point.length * sizeof(State));
    for (int i = 0; i < state_point.length; i++) {
        state_point.state_p[i].hit = 0;
        state_point.state_p[i].miss = 0;
        state_point.state_p[i].eviction = 0;
    }
    return state_point;
}

Cache** initCache(int S, int E) {
    Cache** cache = (Cache **)malloc(S * sizeof(Cache*));
    for (int i = 0; i < S; i++) {
        cache[i] = (Cache *)malloc(E * sizeof(Cache));
    }
    for (int i = 0; i < S; i++) {
        for (int j = 0; j < E; j++) {
            cache[i][j].valid = 0;
            cache[i][j].tag = 0;
        }
    }
    return cache;
}

int** initLRU(int S, int E) {
    int** LRU = (int **)malloc(S * sizeof(int*));
    for (int i = 0; i < S; i++) {
        LRU[i] = (int *)malloc(E * sizeof(int));
    }
    for (int i = 0; i < S; i++) {
        for (int j = 0; j < E; j++) {
            LRU[i][j] = j;
        }
    }
    return LRU;
}

unsigned getBlock(unsigned address, int b) {
    unsigned mask_block = (1 << b) - 1;
```

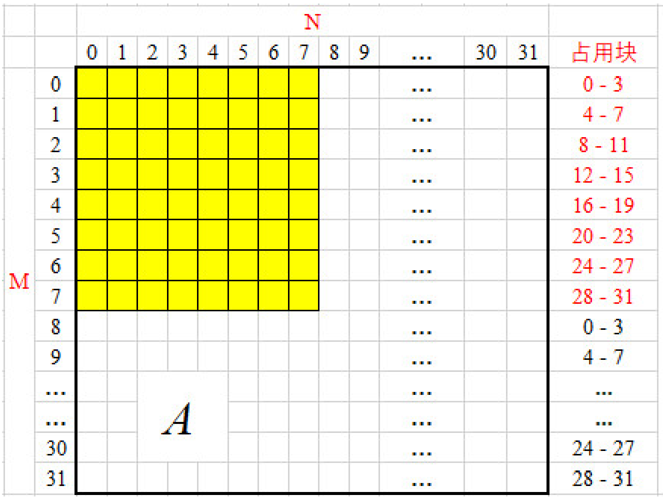
```c
    return (address & mask_block);
}
unsigned getSet(unsigned address, int b, int s) {
    unsigned mask_set = (1 << s) - 1;
    return ((address >> b) & mask_set);
}
unsigned getTag(unsigned address, int b, int s) {
    return (address >> (b + s));
}

void reorderLRU(int** LRU, unsigned set, int val) {
    int id;
    for (id = 0; ;id++){
        if (LRU[set][id] == val){
            break;
        }
    }
    int tmp = LRU[set][id];
    for (int i = id; i > 0; i--) {
        LRU[set][i] = LRU[set][i - 1];
    }
    LRU[set][0] = tmp;
}

//void printSummary(int hit_count, int miss_count, int eviction_count) {
//  printf("hits:%d misses:%d evictions:%d\n", hit_count, miss_count,
eviction_count);
//}

void evalState(InstructPoint instruct_point, Cache** cache_p, int** LRU, int s,
int E, int b, StatePoint state_point) {
    char identifier;
    unsigned address;
    //int size;

    unsigned set;
    unsigned tag;

    int k;
    for (int i = 0; i < instruct_point.length; i++) {
        identifier = instruct_point.instruct_p[i].identifier;
        address = instruct_point.instruct_p[i].address;
        //size = instruct_point.instruct_p[i].size;
        switch (identifier) {
        case 'I':
            break;
        case 'L':
        case 'S':
        case 'M':
            set = getSet(address, b, s);
            tag = getTag(address, b, s);
            for (k = 0; k < E; k++) {
                if (cache_p[set][k].valid && (cache_p[set][k].tag == tag)) {
                    state_point.state_p[i].hit = 1;
                    if (identifier == 'M') {
                        state_point.state_p[i].hit = 2;
                    }
                    reorderLRU(LRU, set, k);
                    break;
                }
            }
            if (k == E) {
                state_point.state_p[i].miss = 1;
                if (identifier == 'M') {
                    state_point.state_p[i].hit = 1;
                }
                cache_p[set][LRU[set][E - 1]].tag = tag;
                if (cache_p[set][LRU[set][E - 1]].valid) {
                    state_point.state_p[i].eviction = 1;
                }
                else {
                    cache_p[set][LRU[set][E - 1]].valid = 1;
                }
                reorderLRU(LRU, set,LRU[set][E - 1]);
            }
            break;
        default:
            printf("Illegal Instruction!!");
            break;
        }
    }
}
```

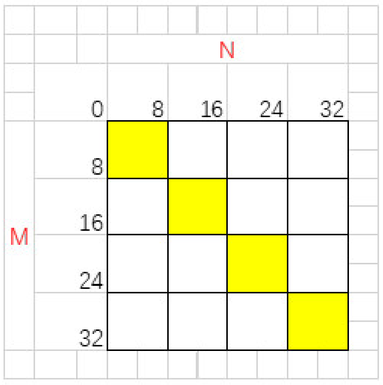
Part B: Optimizing Matrix Transpose

该部分要求将矩阵 *A* 的转置存储在矩阵 *B* 中，并尽可能使缓存不命中次数最少。规定的缓存参数为：*s = 5, E = 1, b = 5*， 则该缓存区有 *S = 32* 个高速缓存组，每个高速缓存组有 *E = 1* 个高速缓存行，共有 *S * E = 32* 个高速缓存块，每个高速缓存块可以存储 *B = 32* 个字节，即可以存储 *8* 个 *int* 类型数字。

该问题主要采用矩阵分块的方法求解。

- 对于维度为 *32 × 32 (M = 32, N = 32)* 的矩阵，易知缓存区域最多可以存储其 *8* 行数据，所以可将矩阵分成维度为 *8 × 8* 分别进行转置。





程序代码如下：

```
void transpose_submit_1(int M, int N, int A[N][M], int B[M][N])
{
    int bsize = 8;
    int i, j, m, n;
    for (i = 0; i < N; i += bsize){
        for (j = 0; j < M; j += bsize){
            for (m = j; m < j+bsize; m++){
                for (n = i; n < i+bsize; n++){
                    B[m][n] = A[n][m];
                }
            }
        }
    }
}
```

矩阵分块后对角线区域（黄色阴影部分）*miss* 次数为：*(9 × 1 + 4 × 7) × 4 = 148* 次，其余部分 *miss* 次数为：*16 × 12 = 192* 次，所以，总的 *miss* 次数为 *340* 。运行结果如下：

> ```
> Step 1: Validating and generating memory traces
> Step 2: Evaluating performance (s=5, E=1, b=5)
> func 1 (Transpose submission_1): hits:1710, misses:343,
> evictions:311
> ```

运行结果为 *343* 次，与估计的相差不大，但题目要求小于 *300* 次，应继续改进。可以看出对角线上分块矩阵每个 *miss* 次数为 *37* 次，主要在于此时矩阵 *A* 和矩阵 *B* 的元素含有重复块。采用对对角线上分块矩阵每一列进行处理时，首先赋值对角线上元素，然后赋值其它元素，可明显减少 *miss* 为：*(9 × 1 + 2 × 7) × 4 = 92* 次，其余部分 *miss* 次数不变，共 *284* 次。代码如下：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int bsize = 8;
    int i, j, m, n;
    int count = 0;
    for (i = 0; i < N-1; i += bsize){
        for (j = 0; j < M-1; j += bsize){
            for (m = j; m < j+bsize; m++){
                if (count%5 == 0){
                    B[m][m] = A[m][m];
                }
                for (n = i; n < i+bsize; n++){
                    if (m != n){
                        B[m][n] = A[n][m];
                    }
                }
            }
            count++;
        }
    }
}
```

运行结果如下：

> ```
> Step 1: Validating and generating memory traces
> Step 2: Evaluating performance (s=5, E=1, b=5)
> func 0 (Transpose submission): hits:1766, misses:287, evictions:255
> ```

输出结果为 *287* 次，满足题目要求，与估计相差不大。当然也可以采用网上大多数引进 *8* 个局部变量的方法减小 *miss* 次数。

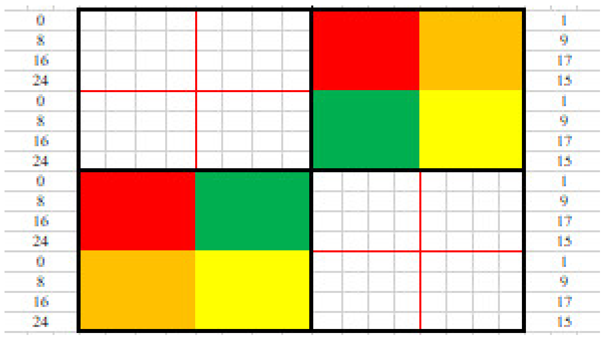
- 对于维度为 *64 × 64 (M = 64, N = 64)* 的矩阵，易知缓存区域最多可以存储其 *4* 行数据，所以可将矩阵分成维度为 *4 × 4* 分别进行转置。代码如下：

```
void transpose_submit_6(int M, int N, int A[N][M], int B[M][N])
{
    int bsize = 4;
    int i, j, m, n;
    for (i = 0; i < N-1; i += bsize){
        for (j = 0; j < M-1; j += bsize){
            for (m = j; m < j+bsize; m++){
                for (n = i; n < i+bsize; n++){
                    B[m][n] = A[n][m];
                }
            }
        }
    }
}
```

输出结果为:

```
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 6 (Transpose submission_6): hits:6354, misses:1843,
evictions:1811
```

题目要求 *miss* 次数应小于 *1300*，尝试将矩阵按照 *8 × 8* 分块，每个块再按照 *4 × 4* 分块，如下图所示，对于 *4* 个 *4 × 4* 小分块按照按照"红-橙-黄-绿"的顺序进行转置。



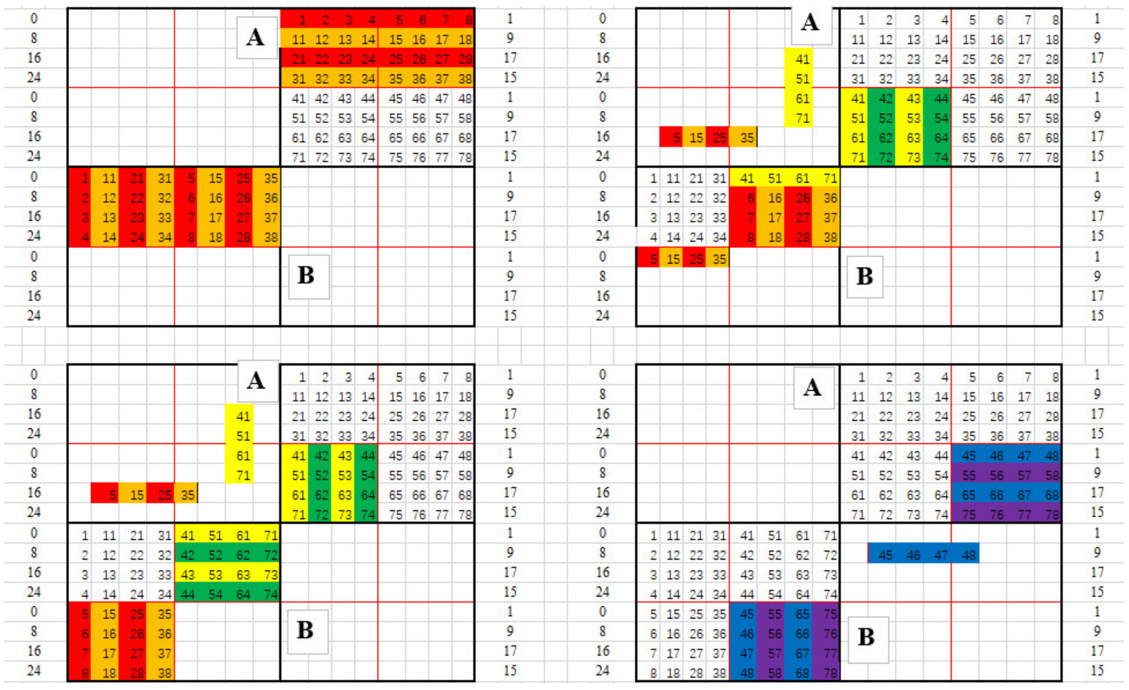
代码如下:

```
void transpose_submit_2(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k, m, n;
    int bsize, Bsize;
    bsize = 4;
    Bsize = 8;
    for (i = 0; i < N; i += Bsize){
       for (j = 0; j < M; j += Bsize){
           for (k = 0; k < 4; k++){
                switch (k){
                    case 0:
                        for (n = i; n < i+bsize; n++){
                            for (m = j; m < j+bsize; m++){
                                B[m][n] = A[n][m];
                            }
                        }
                        break;
                    case 1:
                        for (n = i; n < i+bsize; n++){
                            for (m = j+bsize; m < j+Bsize; m++){
                                B[m][n] = A[n][m];
                            }
                        }
                        break;
                    case 2:
                        for (n = i+bsize; n < i+Bsize; n++){
                            for (m = j+bsize; m < j+Bsize; m++){
                                B[m][n] = A[n][m];
                            }
                        }
                        break;
                    case 3:
                        for (n = i+bsize; n < i+Bsize; n++){
                            for (m = j; m < j+bsize; m++){
                                B[m][n] = A[n][m];
                            }
                        }
                        break;
                    default:
                        break;
                }
            }
        }
    }
}
```

输出结果为：

```
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (Transpose submission_2): hits:6554, misses:1643,
evictions:1611
```

效果稍微好了一点，但还是远远未达到题目要求，原因应是每个缓存块保存着 *8* 个数据，但我们只用了其中 *4* 个，未能有效使用缓存。考虑到题目虽然要求不能对矩阵 *A* 进行修改，但我们可以对矩阵 *B* 进行任意操作，所以，可将 *A* 中每个缓存块的数据一次性赋值到 *B* 中，然后对矩阵 *B* 进行操作，使各个数处于正确的位置。



如上图所示，首先，将矩阵 *A* 的上半部分转置赋值到 *B* 的上半部分，其中，*B* 的左上角数据是正确转置的；然后，依次将 *A* 的左下角的每一列赋值到中间变量，将 *B* 的右上角的每一行赋值到中间变量，将前四个值赋值到 *B* 的右上角的每一行，将后四个值赋值到 *B* 的左下角的每一行，此时 *B* 的数据除右下角外都是正确转置的；最后，依次取 *A* 的右下角的每一行赋值给中间变量，再将中间变量赋值给 *B* 的右下角的每一列。程序如下：

```
void transpose_submit_4(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, m, n;
    int val0, val1, val2, val3, val4, val5, val6, val7;
    for (i = 0; i < N; i += 8){
        for (j = 0; j < M; j += 8){
            m = j;
            for (n = i; n < i+4; n++){
                val0 = A[n][m];
                val1 = A[n][m+1];
                val2 = A[n][m+2];
                val3 = A[n][m+3];
                val4 = A[n][m+4];
                val5 = A[n][m+5];
                val6 = A[n][m+6];
                val7 = A[n][m+7];

                B[m][n] = val0;
                B[m+1][n] = val1;
                B[m+2][n] = val2;
                B[m+3][n] = val3;
                B[m][n+4] = val4;
                B[m + 1][n+4] = val5;
                B[m + 2][n+4] = val6;
                B[m + 3][n+4] = val7;
            }
            n = i+4;
            for (m = j; m < j+4; m++){
                val0 = A[n][m];
                val1 = A[n+1][m];
                val2 = A[n+2][m];
                val3 = A[n+3][m];

                val4 = B[m][n];
                val5 = B[m][n+1];
                val6 = B[m][n+2];
                val7 = B[m][n+3];

                B[m][n] = val0;
                B[m][n+1] = val1;
                B[m][n+2] = val2;
                B[m][n+3] = val3;

                B[m+4][i] = val4;
                B[m+4][i+1] = val5;
                B[m+4][i+2] = val6;
                B[m+4][i+3] = val7;
            }
            m = j + 4;
            for (n = i + 4; n < i + 8; n++) {
                val0 = A[n][m];
                val1 = A[n][m+1];
                val2 = A[n][m+2];
                val3 = A[n][m+3];

                B[m][n] = val0;
                B[m+1][n] = val1;
                B[m+2][n] = val2;
                B[m+3][n] = val3;
            }
        }
    }
}
```

输出结果为：

```
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 4 (Transpose submission_4): hits:9066, misses:1179,
evictions:1147
```