

天津大学

计算机系统基础上机实验报告

实验题目 5：高速缓存 cache

学院名称_____未来技术学院_____

专 业_____计算机科学与技术_____

学生姓名_____石云天_____

学 号_____3020205015_____

年 级_____2020 级_____

班 级_____2 班_____

时 间_____2022 年 12 月 14 日_____

实验 5：高速缓存

Cache

1. 实验目的

进一步理解高速缓存对于程序性能的影响。

2. 实验内容

这个实验包括以下两部分内容：你需要使用 C 语言编写一个小型程序（200-300 行）用来模拟高速缓存；然后，对一个矩阵转置函数进行优化，以减少函数操作中的缓存未命中次数。详细内容请参考实验指导书：实验 5.pdf

No.	任务内容
1	任务 A：编写一个高速缓存模拟程序。在这部分任务中，你将在 <code>csim.c</code> 文件中编写一个高速缓存仿真程序。这个程序使用 <code>valgrind</code> 的内存跟踪记录 作为输入，模拟高速缓存的命中/未命中行为，然后输出总的命中次数，未命中次数和缓存块的替换次数。
2	任务 B：优化矩阵转置运算程序。在 <code>trans.c</code> 中编写一个矩阵转置函数，尽可能的减少程序对高速缓存访问的未命中次数。
3	

3. 实验要求

- 1) 在 Ubuntu18.04 LTS 操作系统下，按照实验指导说明书，使用 `gcc`、`make` 和内存访问进行捕获和追踪的工具，完成本实验。
- 2) 本实验的具体要求：
 - a) 编译时不允许出现任何的 `warning`。
 - b) 转置函数中定义的 `int` 型局部变量总数不能超过 12 个。
 - c) 不允许使用 `long` 等数据类型，在一个变量中存储多个数组元素以减

少内存访问。

d) 不允许使用递归。

e) 在程序中不能修改矩阵 A 中的内容，但是，你可以任意使用矩阵 B 中的空间，只要保证最终的结果正确 即可。

f) 在函数中不能定义任何的数组，不能使用 malloc 分配额外的空间。

3) 需提交: csim.c 和 trans.c 源文件，电子版实验报告全文。

4. 实验目的

(给出程序实现思路、编码风格、测试结果截图等)

4.1 Cache 部分

(1) 输入处理:

通过观察仿真器的输入，发现需要先对输入参数进行处理，利用 getopt 函数处理传入的参数。其中 -s -E -b -t 为测试时的必要输入，而 -h, -v 为可选输入。输入 -h 则打印帮助信息，输入 -v 则在模拟过程中输出跟踪信息，于是定义全局变量 verbose 来作为 -v 参数的 flag，仿真器具体使用方法见下图 1:

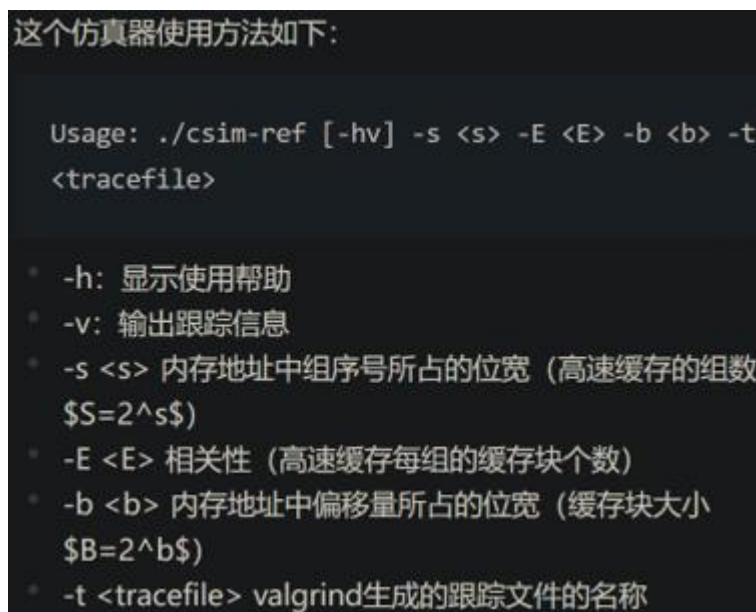


图 1 仿真器具体使用方法

随后编写输入处理代码:

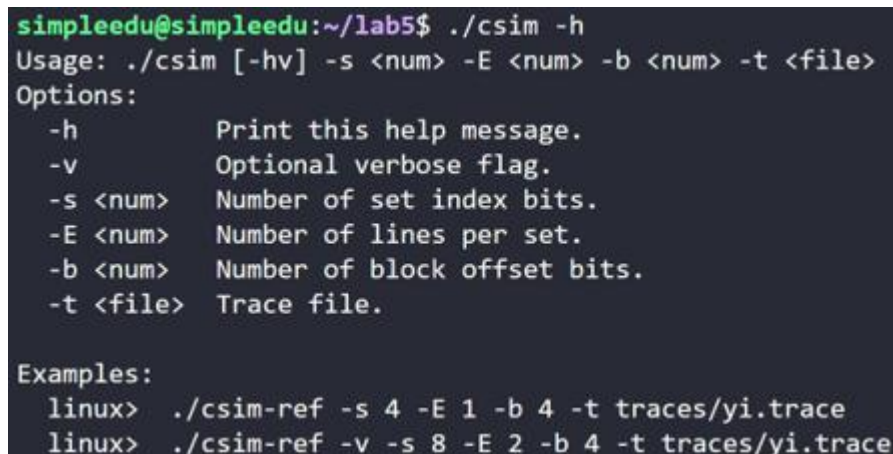
```
int hit_count = 0, miss_count = 0, eviction_count = 0; // 输出结果
int s, E, b; // cache 的参数
long long _Tag, _Index;
int o; // 读入命令行参数
```

```

FILE *fp;// 读入文件句柄
// 处理输入
while ( ( o = getopt( argc, argv, "hvs:E:b:t:" ) ) != -1 )
{
    switch ( o )
    {
        case 'h': Printh( argv[ 0 ] ); break;
        case 'v': break;
        case 's': s = *optarg - '0'; break;
        case 'E': E = *optarg - '0'; break;
        case 'b': b = *optarg - '0'; break;
        case 't':
            if ( ( fp = fopen( optarg, "r" ) ) == NULL )
            {
                printf( "cannot open this file\n" );
                exit( 0 );
            }
            break;
        default:
            printf( "Invalid option '%c'\n", o );
            Printh( argv[ 0 ] );
            break;
    }
}

```

输入测试指令“`./csim -h`”, 所得结果见下图 2:



```

simpleedu@simpleedu:~/lab5$ ./csim -h
Usage: ./csim [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
  -h          Print this help message.
  -v          Optional verbose flag.
  -s <num>    Number of set index bits.
  -E <num>    Number of lines per set.
  -b <num>    Number of block offset bits.
  -t <file>   Trace file.

Examples:
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
linux> ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace

```

图 2 测试结果

由于输出函数已经给出, 使用 PrintSummary, 不需要自己实现, 因此将 verbose 置为废弃。

(2) Cache 定义:

首先考虑 Cache 的结构, 一个 Cache 中含有多个组, 一个组中含有多个 line, 一个 line 存放一个 block 和一些标志位, 由于本次模拟只关注命中、不命中和替换数, 故定义时无需考虑到整个 block 保存的过程。

接下来定义 line 结构，因为此处其不具有存储数据的功能，故只定义标志作为属性。定义 val 属性用来模拟有效位。定义 Tag 属性用于模拟 Tag 位，考虑到地址为 64 位二进制数，极端情况下全相联组数为 1 且块大小为 1 字节时，地址中 64 位将全部属于 Tag，因此定义的 Tag 属性应至少有 64 位，于是变量类型定义为 longlong。定义 LRU 属性用于模拟 LRU，LRU 的取值范围取决于关联度，利用给出的测试程序进行测试，发现即使在全相联的情况下当关联度达到 429496726（小于 2 的 32 次方）时便会发生段错误（见下图 3），因此 int 类型便足以表示全部情况。本次模拟中无需定义 dirty 位，是因为 dirty 位用于块替换的过程，在本次模拟过程中，dirty 不会对结果造成任何影响。

```
simpleedu@simpleedu:~/lab5$ ./csim-ref -s 1 -E 429496726 -b 1 -t ./traces/yi.trace
段错误
```

图 3 关联度为 429496726 时测试结果

最后定义 Cache 结构，其包括 s（组序号所占位宽），E（组相联度），b（偏移量位宽），t（tag 位宽）以及指向 line 的 lines 指针。s、E、b、t 都是 Cache 的属性，lines 表示 cache 的行，具体见下方代码：

// 定义 line，其中 dirty 位用于判断块替换时，是否将 cache 的内容写回主存中，对本次模拟不造成任何影响，因此不维护 dirty 位

```
struct line
{
    char val; //V 位 只需要 1bit 即可，只有 0 和 1 两种状态，用最小字节的变量类型 char
    long long Tag; //Tag 位，由于实验要求中给出地址类型为 64 位，所以极端情况下 Tag 会占到 62 位，故使用 long long 类型
    int LRU; // LRU 位，为支持 LRU 替换策略而添加的位，LRU 位的长度取决于关联度，极端情况下全相联，cache 容量等于主存容量，不会超过 2 的 32 次方，故使用 int 类型
};
```

```
// 定义 cache
struct cache
{
    int s; // 组序号所占位宽
    int E; // 组的相联度
    int b; // 偏移量位宽
    int t; // tag 号所占位宽
    Ptrline lines; // cache 主体部分
};
```

(3) Cache 的初始化：

初始化过程主要分为两步，第一步初始化 Cache，第二步初始化 lines，在初始化 cache 的过程中，需要读入 s、E、b 属性并计算 t 属性。在初始化 lines 的过程中首先需要根据 cache 的 s 和 E 属性计算出 line 的个数，再将每一个 line 的 val、

Tag 和 LRU 属性置零，其中 val 置零必需，因为没有用到的 line 的 val 默认是 0，Tag 和 LRU 置零与否皆可，具体代码如下：

```
Ptrcache CreateCache( int s, int E, int b )
{
    // 实体化 cache
    Ptrcache Cache = ( Ptrcache )malloc( sizeof( struct cache ) );
    Cache->s = s;
    Cache->E = E;
    Cache->b = b;
    Cache->t = 64 - s - b;
    int line_num = ( 1 << Cache->s ) * Cache->E;
    Cache->lines = ( Ptrline )malloc( sizeof( struct line ) * line_num );
    for ( int i = 0; i < line_num; i++ )
    {
        Cache->lines[ i ].val = 0;
        Cache->lines[ i ].Tag = 0;
        Cache->lines[ i ].LRU = 0;
    }
    return Cache;
}
```

(4) 文件输入处理：

利用 fgets 读入每一行的数据，只有开头为空格的行才是有效行，在读入每一行的同时，得到操作 op、地址 addr 和读入数据的 size，随后针对不同的操作执行不同的模拟过程。此外还需要对读入地址进行解析，地址由以下三部分组成：Tag、Index、offset，见下图 4：



图 4 地址组成

我们可以根据 Cache 的 b 属性来提取 offset 区域，直接取低 b 位即可，接下来根据 b 和 s 属性提取 Index 区域，只需把低 b 位去除即可，在对去除后的低 s 位进行掩码操作即可。提取 Tag 属性的操作与提取 Index 的操作类似，具体见下方代码：

```
while ( fgets( buf, 100, fp ) != NULL )
{
    if ( buf[ 0 ] == ' ' )
    {
```

```

        sscanf( buf, " %c %llx,%d", &op, &addr, &size );
        // 提取 Tag 区域和 Index 区域
        _Tag = ( ( addr >> ( ( C->b ) + ( C->s ) ) ) & ~( -1 << C->t ) );
        _Index = ( ( addr >> ( C->b ) ) & ( ( 1 << ( C->s ) ) - 1 ) );
        switch ( op )
        {
            case 'I': break;
            case 'L':
            case 'S':
                LoadOrSaveCache(C,_Tag,_Index,&hit_count,
                &miss_count, &eviction_count );
                break;
            case 'M':
                ModifyCache(C,_Tag, _Index, &hit_count, &miss_count,
                &eviction_count );
                break;
        }
    }
}

```

(5) 加载模拟:

加载模拟过程主要分为三步: 1.定位组; 2.判断命中; 3.未命中时的替换。定位组的过程利用提取到的 Index 信息以及 Cache 的关联度, 来确定是哪一组以及该组对应 line 的范围。其中提取得到的 Index 信息即为组数, 随后利用 Index*E 得到该组的第一行 line_index_first, 利用 line_index_first+E 得到该组的最后一行的下一行 line_index_end (line_index_end 不属于这组), 具体代码如下:

```

void Index2LineRange( int *line_index_first,
                    int *line_index_end,
                    int E,
                    int Index )
{
    // 组的关联度为 E, 计算第 Index 组所对应 line 的范围
    *line_index_first = Index * E;
    *line_index_end   = *line_index_first + E;
}

```

判断命中的过程是对组内所有有效行的 Tag 进行比较, 如果某一行 val 为 1 且 Tag 也与输入的 Tag 相同, 则命中, 命中数加 1。命中后需要对 LRU 进行更

新，将组内所有 LRU 小于命中 line 的 LRU 都+1，并把命中 line 的 LRU 置零，具体过程见下方代码：

```
int TagHitLineRange( Ptrcache C,
                    int      line_index_first,
                    int      line_index_end,
                    long long Tag )
{
    // 对组内所有 line 比较 Tag 判断是否命中
    for ( int line = line_index_first; line < line_index_end; line++ )
    {
        // Tag 命中
        if ( C->lines[ line ].val == 1 && C->lines[ line ].Tag == Tag )
        {
            // 修改组内其他 LRU 位
            LRUUpdateLineRange( C, line_index_first, line_index_end,
                                C->lines[ line ].LRU );
            C->lines[ line ].LRU = 0; // hit 后 LRU 置零
            return 1;
        }
    }
    return 0;
}
```

如果未命中的需要执行替换策略，替换的时候分为两种情况，第一种是组内还有 val 为 0 的行(即没有存放数据的 line)，这种情况直接把其他所有行 LRU+1，并把该行 LRU 置零，此时 miss 数+1，替换数不变。第二种是需要选择 LRU 最大的行进行替换，替换后把所有小于 LRU 的行+1，并把该行 LRU 置零，此时替换数和命中数都+1。在该过程中，需要先去遍历寻找 val 为 0 的行，再去遍历寻找 LRU 最大的行，为减少时间复杂度，将这两步放到函数 GetVallsZeroAndMax LRULineIndex 中一起进行，如果函数返回值为 0 则说明没有 val 为 0 的 line，具体代码如下：

```
{
    // 如果没命中，进行替换策略
    int line_index_val_is_zero = line_index_first,
        line_index_max_LRU     = line_index_first;
    ( *miss_count )++;
    // 不存在 val=0 的 line，根据 LRU 进行替换
```



```

        if ( GetValIsZeroAndMaxLRULineIndex(C, line_index_first,
line_index_end, &line_index_val_is_zero,&line_index_max_LRU ) == 0 )
        {
            C->lines[ line_index_max_LRU ].Tag = Tag;
            LRUUpdateLineRange( C, line_index_first, line_index_end,
                C->lines[ line_index_max_LRU ].LRU );
            C->lines[ line_index_max_LRU ].LRU = 0; // hit 后 LRU 置零
            ( *eviction_count )++;
        }
        else
        {
            // 存在 val = 0 的 line, 根据 val 进行放入
            C->lines[ line_index_val_is_zero ].val = 1;
            C->lines[ line_index_val_is_zero ].Tag = Tag;
            LRUUpdateLineRange(C, line_index_first, line_index_end,
                ( C->E + 1 ) ); // 放入 val=0 的 line, 把所有 LRU 加一
            C->lines[ line_index_val_is_zero ].LRU = 0;
        }
    }
}

```

LRU 的更新策略如下: 遍历 line 找到所有 val 为 1 且 LRU 小于输入 LRU 的 line, 将它们的 LRU+1, 且在此过程中维护 LRU 的最大值, 当 LRU 已经达到最大值时不会进行+1 操作。故 val 为 0 时输入的 LRU 应为 LRU 的最大值 E, 具体代码如下:

```

void LRUUpdateLineRange( Ptrcache C,
                        int      line_index_first,
                        int      line_index_end,
                        int      LRU )
{
    // 对 LRU 小于输入 LRU 的 line 进行更新
    for ( int line = line_index_first; line < line_index_end; line++ )
    {
        if ( C->lines[ line ].val == 1 && C->lines[ line ].LRU < LRU )
            // 找到组内 LRU 值小于该 line 的 line
            {
                if ( C->lines[ line ].LRU < C->E ) // 维护 LRU 的最大值
                {

```

```

        C->lines[ line ].LRU++; // LRU 增大
    }
    // 若 LRU 以达到最大则不动
}
}
}
}

```

(6) 保存与修改模拟:

保存过程在本次实验中对结果的影响与加载过程并无任何不同,因此保存与加载共用一个模拟函数即可,而修改过程本质上是一次加载和保存,故修改过程只需调用两次加载模拟函数即可,具体代码如下:

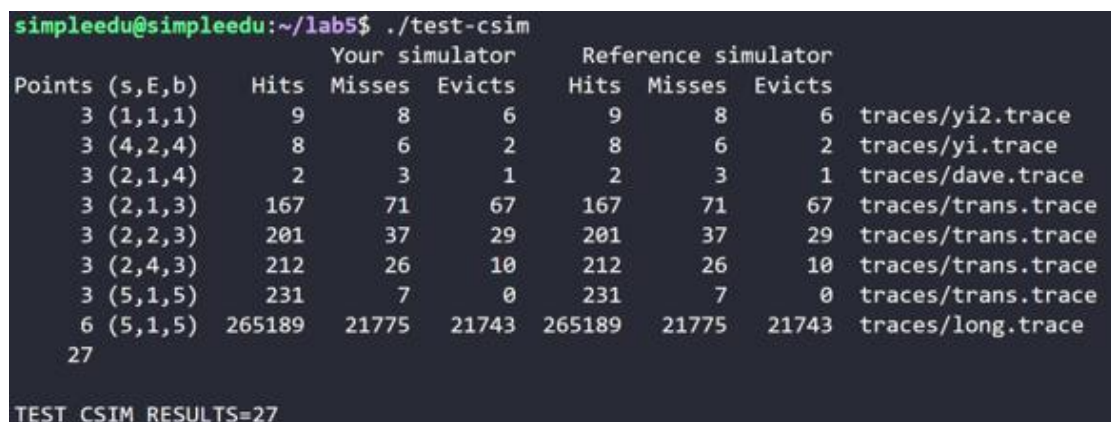
```

void ModifyCache( Ptrcache C,
                  long long Tag,
                  int      Index,
                  int      *hit_count,
                  int      *miss_count,
                  int      *eviction_count )
{
    // modify 本质上是先做一次 Load 再做一次 Save
    LoadOrSaveCache( C, Tag, Index, hit_count, miss_count,
eviction_count );
    LoadOrSaveCache( C, Tag, Index, hit_count, miss_count,
eviction_count );
}

```

(7) 测试结果:

利用指令 “./test-csim” 进行测试, 所得结果见下图 5:



Points (s,E,b)	Your simulator			Reference simulator			
	Hits	Misses	Evicts	Hits	Misses	Evicts	
3 (1,1,1)	9	8	6	9	8	6	traces/yi2.trace
3 (4,2,4)	8	6	2	8	6	2	traces/yi.trace
3 (2,1,4)	2	3	1	2	3	1	traces/dave.trace
3 (2,1,3)	167	71	67	167	71	67	traces/trans.trace
3 (2,2,3)	201	37	29	201	37	29	traces/trans.trace
3 (2,4,3)	212	26	10	212	26	10	traces/trans.trace
3 (5,1,5)	231	7	0	231	7	0	traces/trans.trace
6 (5,1,5)	265189	21775	21743	265189	21775	21743	traces/long.trace

27

TEST CSIM RESULTS=27

图 5 Cache 部分测试结果

4.2 Transpose 部分

(1) is_transpose 分析:

首先分析给出的样例代码，具体代码如下：

```
int is_transpose(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; ++j) {
            if (A[i][j] != B[j][i]) {
                return 0;
            }
        }
    }
    return 1;
}
```

分析代码可知：其会按行优先读取 A 矩阵，然后逐列写入 B 矩阵。已知 C 语言中，矩阵是按行排列的，即每行的内容在内存中是连续的，相邻行的内存是连续的。已知缓存每次会从内存中加载固定大小的内存块。故该方法读取 A 矩阵对缓存是友好的，但对 B 矩阵的存储是逐列进行的，因此在极端情况下，每次访问 B 矩阵都会造成缓存不命中。为优化性能，使用分块的思想编写缓存友好代码。

(2) 32 * 32 矩阵转置:

由题目知共有 32 个缓存块，每块大小为 32bytes，可存放 8 个 int 型数据，故 32×32 矩阵每行需要 4 个缓存块，相隔 8 行会造成缓存块的冲突，见下图 6：

0		0		1		2		3	
1		4		5		6		7	
2		8		9		10		11	
3		12		13		14		15	
4		16		17		18		19	
5		20		21		22		23	
6		24		25		26		27	
7		28		29		30		31	
8		0		1		2		3	

图 6 32×32 矩阵冲突情况

因此我们定义分块大小为 8×8，编写转置代码如下：

```
for ( i = 0; i < N; i += 8 )
{
    for ( j = 0; j < M; j += 8 )
    {
```

```

// 分块
for ( k = i; k < min( i + 8, N ); k++ )
{
    for ( s = j; s < min( j + 8, M ); s++ )
    {
        B[ s ][ k ] = A[ k ][ s ];
    }
}
}

```

经测试发现 miss 次数达到了 343 次，对代码进行分析，每个分块的大小为 8×8 ，所以每个块的 miss 次数是 8。每个矩阵有 16 个分块，有两个矩阵，所以总的 miss 次数就是 $8 \times 16 \times 2 = 256$ ，分析结果与实际结果相差巨大。输出 A 和 B 矩阵的地址，可以发现两者地址低 16 位相同，而 Index 是低 5-10 位组成的，故 A 和 B 在缓存中会发生冲突，由于 A 和 B 矩阵互为转置，故冲突只发生在对角线元素上，即读入块 A 保存有 A 的对角线元素该行，此时块 B 还未读入，在转置时读入块 B，转置对角线元素时所在行也会读入，两者对应 Index 相同，此时块 B 对角线元素所在行将块 A 对角线元素所在行取代，替换完对角线元素后，继续进行转置会重新将块 A 该行读入，这样会产生 2 次多余的 miss。第一行和最后一行有所不同只会，其产生 1 次额外的 miss。等到下一次替换 B 的对角线元素所在行时会重新读入，产生 1 次额外的 miss。对角块有 8 个对角元素，总共产生额外 miss 的次数为 $6 \times 2 + 2 \times 1 + 7 = 21$ 次。因此分析可得总 miss 次数应为： $8 \times 16 \times 2 + 21 \times 4 = 340$ 次，与实际结果相同，额外的 3 次 miss 为函数的开销。

考虑到上述分析情况，为避免对角线上产生的冲突，利用局部变量存储 A 矩阵块的一行所有元素，再复制给 B 矩阵，局部变量数目不多存放在寄存器上，可以减小开销，具体代码如下：

```

for (i = 0; i < 32; i += 8)
{
    for (j = 0; j < 32; j += 8)
    {
        for (k = i; k < i + 8; k++)
        {
            a0 = A[k][j];
            a1 = A[k][j + 1];
            a2 = A[k][j + 2];
            a3 = A[k][j + 3];

```

```

        a4 = A[k][j + 4];
        a5 = A[k][j + 5];
        a6 = A[k][j + 6];
        a7 = A[k][j + 7];
        B[j][k] = a0;
        B[j + 1][k] = a1;
        B[j + 2][k] = a2;
        B[j + 3][k] = a3;
        B[j + 4][k] = a4;
        B[j + 5][k] = a5;
        B[j + 6][k] = a6;
        B[j + 7][k] = a7;
    }
}
}

```

分析该段代码，此时对角线块只剩下复制 A 块第 m 行时会覆盖掉 B 块第 m 行，因此每个对角线块有额外的 7 次 miss，故总 miss 为 $8*16*2+7*4=284$ 次，这与测试结果 287 次相符。为消除对角线块上的分行的额外开销，可以先将 A 块内容直接不转置地存入 B 块中，再在 B 块中进行转置，这样就可以避免冲突，达到理论最优解。miss 次数为 $8*16*2=256$ 次，测试结果 miss 次数为 259 次，不过这种方法会带来时间上的开销，是否真正节省时间还要根据实际情况分析，具体代码如下：

```

{
    for ( i = 0; i < N; i += 8 )
    {
        for ( j = 0; j < N; j += 8 )
        {
            // 复制过程
            for ( k = i, s = j; k < i + 8; k++, s++ )
            {
                a0 = A[ k ][ j ];
                a1 = A[ k ][ j + 1 ];
                a2 = A[ k ][ j + 2 ];
                a3 = A[ k ][ j + 3 ];
                a4 = A[ k ][ j + 4 ];
                a5 = A[ k ][ j + 5 ];
            }
        }
    }
}

```


(3) 64 * 64 矩阵转置:

首先分析每一个区块所映射到的缓存单元, 见下图 8:

0	7 8	15 16	23 24	31 32	39 40	47 48	55 56	63
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	
Line 00	Line 01	Line 02	Line 03	Line 04	Line 05	Line 06	Line 07	
Line 08	Line 09	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	
Line 16	Line 17	Line 18	Line 19	Line 20	Line 21	Line 22	Line 23	
Line 24	Line 25	Line 26	Line 27	Line 28	Line 29	Line 30	Line 31	

图 8 区块映射缓存单元

从中可以发现冲突不光发生在对角线上, 还发生在同一块的上下两部分, 对于非对角线区域, 将一个块均分成 4 份, 如下图所示:

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

图9 非对角线区域分块

首先将 A 块的 1、2 区域按行读入放入 B 块的 1、2 区域，放置方法为：A 块 1 区域第一行，放入 B 块 1 区域第一列，A 块 2 区域第一行放入 B 块 2 区域第一列，重复该过程，直到完成 1、2 区域放置，该过程一共出现 8 次 miss，该过程见下图 10：

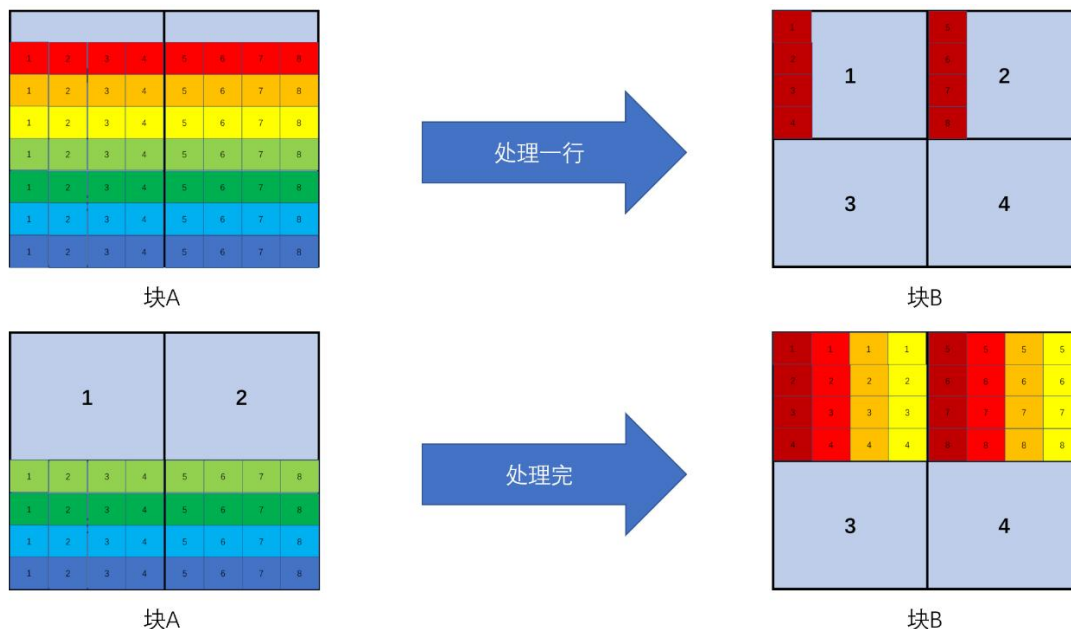


图 10 A 块 12 区域处理过程 miss 分析

再处理 A 块 3 区域，先将 B 块 2 区域第一行放入 B 块 3 区域第一行，再将 A 块 3 区域第一列放入 B 块 2 区域第一行，重复此过程，直至 B 块 3 区域完成，该过程一共出现 8 次 miss，过程见下图 11：

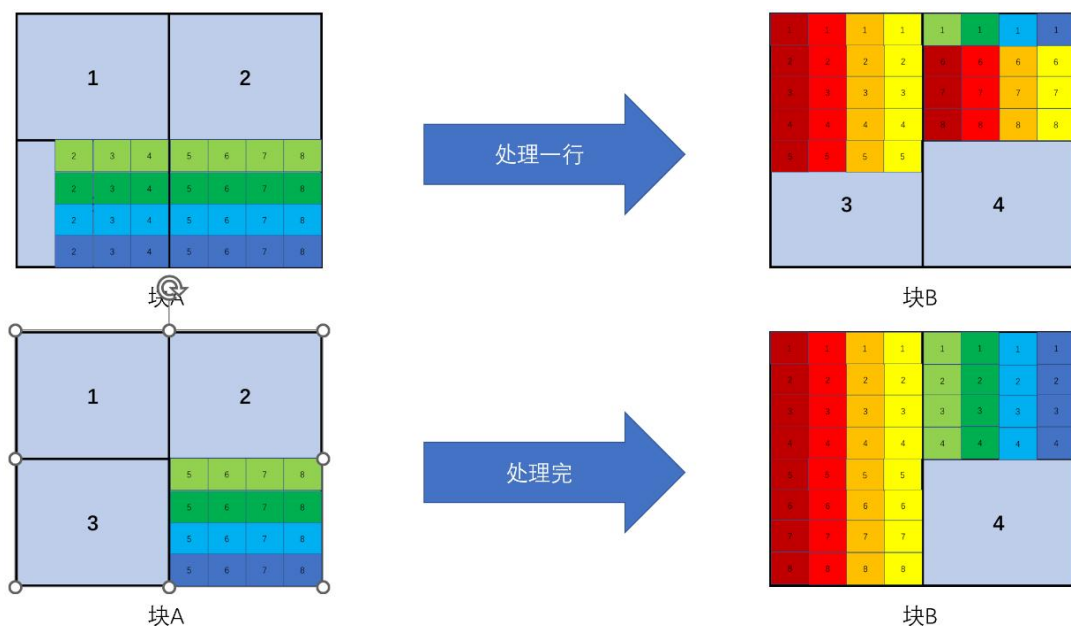


图 11 A 块 3 区域处理过程 miss 分析

最后将 A 块 4 区域放入 B 块 4 区域即可，将第一行放入第一列重复过程即

可，该部分没有 miss，过程见下图 12：

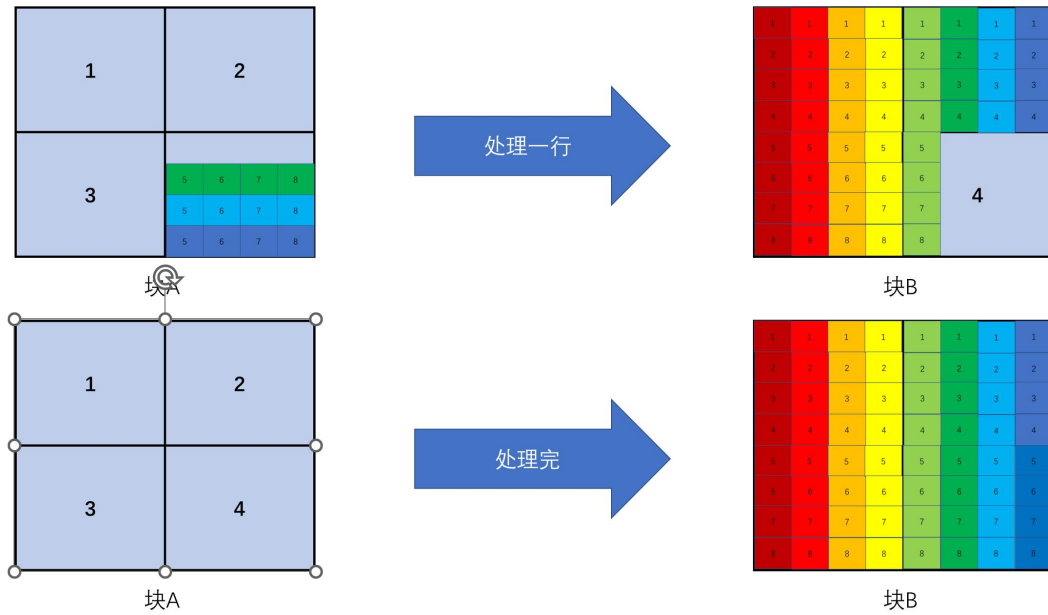


图 12 A 块 4 区域处理过程 miss 分析

该过程完成了 8*8 块的转置且只产生了 16 次 miss，接下来关注对角线块的转置：

对角线块的主要难点在于：A 块 1、2 区域、A 块 3、4 区域、B 块 1、2 区域和 B 块 3、4 区域对应相同的缓存块。为处理对角线块我们需要在 A 块 B 块外引入一个 C 块（非对角块）作为辅助块，C 块的选择是处理完该对角块后 B 矩阵的下一个待处理块，同样把每一块分成四个区域。首先把 A 块的 3、4 区域放置 C 块的 1、2 区域，该过程产生 8 次 miss，再将 A 块的 1、2 区域放入 B 块的 1、2 区域，该过程产生 8 次 miss，处理后 Cache 中存放着 B 块 1、2 区域和 C 块 1、2 区域，结果见下图 13：

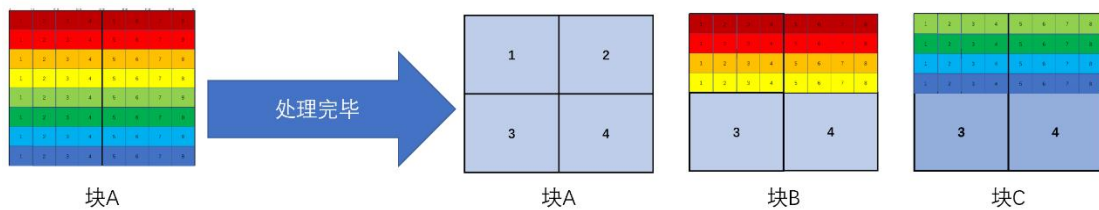


图 13 转置过程演示 I

此过程必须先处理块 A 的 3、4 区域，若先处理 A 块 1、2 区域，会导致该过程结束后 Cache 中存放的数据为 A 块 3、4 区域和 C 块的 1、2 区域，后续处理 B 块 1、2 区域需要重新读入 Cache，造成额外的 4 次 miss。同时必须使用 C 块的 1、2 区域，只有这样才能在对角线块做完之后，Cache 中存放有 C 块的 1、2 区域，避免 C 块的转置时 1、2 区域的首次读入。之后对 B 块 1、2 区域和 C 块 1、2 区域进行转置，该过程不产生 miss，结果见下图 14：

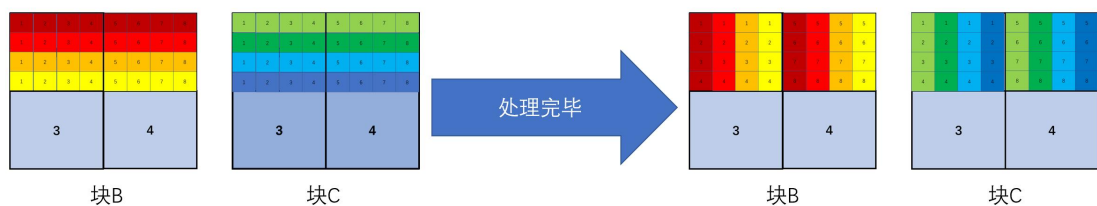


图 14 转置过程演示 II

随后交换 B 块 2 区域和 C 块 1 区域，该过程不产生 miss，结果见下图 15:

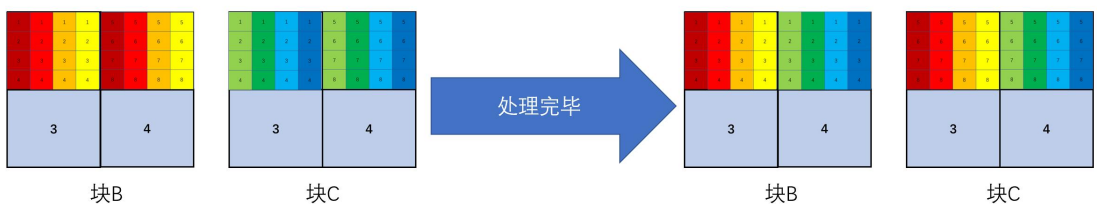


图 15 转置过程演示 III

最后完成 B 块 3、4 区域和 C 块 3、4 区域的交换，该过程产生 4 次 miss，结果见下图 16:

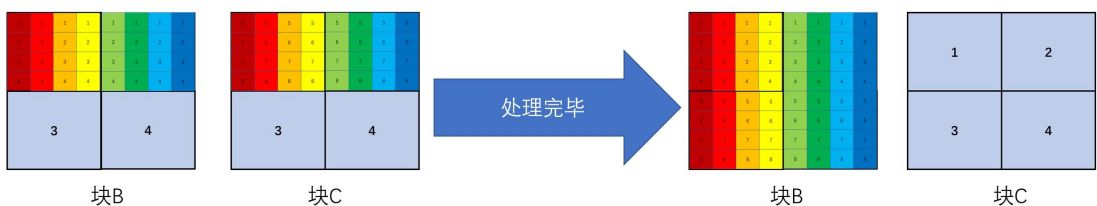


图 16 转置过程演示 IV

该过程总共存在 20 次 miss，但是块 C 将会在接下来立刻被使用，不需要重复读入 C 块 12 区域，故可以节省 4 次 miss。

综上总体过程为：每次先处理对角块，之后再处理第一块，过程和 miss 次数见下图 17:

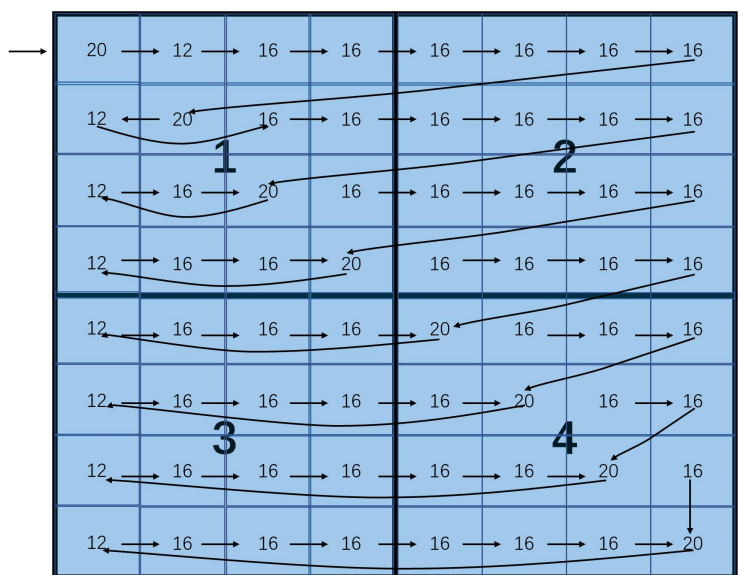


图 17 总体过程演示

miss 次数共计为 1024，为理论最优解，测试结果见下图 18:

```
simpleedu@simpleedu:~/lab5$ ./test-trans -M 64 -N 64

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:10754, misses:1027, evictions:995

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1027

TEST_TRANS_RESULTS=1:1027
```

图 18 64×64 矩阵测试结果

(4) 61×67 矩阵转置:

整体思路按照 32×32 的思路进行，分块转置即可，代码如下:

```
for ( i = 0; i < N; i += 8 )
{
    for ( j = 0; j < M; j += 11 )
    {
        if ( i + 8 <= N && j + 11 <= M )
        {
            for ( s = j; s < j + 11; s++ )
            {
                a0 = A[ i + 0 ][ s ];
                a1 = A[ i + 1 ][ s ];
                a2 = A[ i + 2 ][ s ];
                a3 = A[ i + 3 ][ s ];
                a4 = A[ i + 4 ][ s ];
                a5 = A[ i + 5 ][ s ];
                a6 = A[ i + 6 ][ s ];
                a7 = A[ i + 7 ][ s ];

                B[ s ][ i + 0 ] = a0;
                B[ s ][ i + 1 ] = a1;
                B[ s ][ i + 2 ] = a2;
                B[ s ][ i + 3 ] = a3;
                B[ s ][ i + 4 ] = a4;
                B[ s ][ i + 5 ] = a5;
```

```

        B[ s ][ i + 6 ] = a6;
        B[ s ][ i + 7 ] = a7;
    }
}
else
{
    for ( k = i; k < ( ( i + 8 < N ) ? i + 8 : N ); k++ )
    {
        for ( s = j; s < ( ( j + 11 < M ) ? j + 11 : M ); s++ )
        {
            B[ s ][ k ] = A[ k ][ s ];
        }
    }
}
}
}

```

分块大小定义为 $8 \times x$ ，对 x 进行不同输入测试，测试输入有 11, 13, 17, 23，从中选取效果最好的一种即可，具体测试结果如下：

当 x 选取为 11 时，func0 的 miss 次数为 1798；

当 x 选取为 13 时，func0 的 miss 次数为 1821；

当 x 选取为 17 时，func0 的 miss 次数为 1821；

当 x 选取为 23 时，func0 的 miss 次数为 1863；

综上，最终选择 8×11 的分块。

4.3 结果测试

利用指令“`./driver.py`”进行测试，所得结果见下图 19：

```
simpleedu@simpleedu:~/lab5$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim

      Your simulator      Reference simulator
Points (s,E,b) Hits Misses Evicts Hits Misses Evicts
3 (1,1,1)      9      8      6      9      8      6 traces/yi2.trace
3 (4,2,4)      8      6      2      8      6      2 traces/yi.trace
3 (2,1,4)      2      3      1      2      3      1 traces/dave.trace
3 (2,1,3)     167     71     67     167     71     67 traces/trans.trace
3 (2,2,3)     201     37     29     201     37     29 traces/trans.trace
3 (2,4,3)     212     26     10     212     26     10 traces/trans.trace
3 (5,1,5)     231      7      0     231      7      0 traces/trans.trace
6 (5,1,5)    265189  21775  21743  265189  21775  21743 traces/long.trace
27

Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:

      Points      Max pts      Misses
Csim correctness      27.0      27
Trans perf 32x32       8.0      8      259
Trans perf 64x64       8.0      8     1027
Trans perf 61x67     10.0     10     1798
Total points      53.0     53
```

图 19 实验总体测试结果

从图中可以看出，两部分均为满分，符合实验要求。

5. 实验总结与心得体会

（实验结果总结，实验中遇到的问题及解决方法等）

通过本次拆弹实验，对 Cache 有了更加深刻的理解，对于替换策略以及 LRU 的详细过程掌握的更加深入透彻。在实际设计编写 Cache 的过程中，我进一步了解了 Cache 中的不同标识的作用与 Cache 的命中过程，在 Cache 命中时，LRU 如何更新以及未命中时不同的替换策略与替换时 LRU 如何更新。此外，本次实验中编写的 Cache 需要支持命令行处理，这也要求我们掌握 C 语言处理命令行与文件的输入。在完成第二部分矩阵转置算法设计时，体会到分块思想的重要作用，学会了如何选择分块大小与转置策略以最大程度上实现缓存友好。

在优化 64×64 矩阵转置时，起初设计未能实现实验要求，通过查阅相关资料，发现引入辅助矩阵可以进一步减少 miss 次数，实现理论上最优的情况，解决问题。在编写 LRU 替换策略时也出现了错误，导致在样例测试时出现少了一次 hit 的情况，经过对相关代码的检查，发现我误将所有小于等于输入 LRU 行的 LRU 都加 1，应将此处的小于等于改为小于，经过修改，成功解决问题。