# Cachelab 实验报告

高伟国 S211000806

**实验描述：**

实验包含两个部分。实验 A 需要编写一个缓存模拟器，实验 B 需要针对高速缓存性能优化编写一个矩阵转置功能，以最小化模拟缓存上的未命中数。

实验目录的 traces 子目录包含了所有的参考跟踪文件，用于评估实验 A 中编写的缓存模拟器是否达到实验要求。跟踪文件由 valgrind 程序产生。

**实验 A：编写缓存模拟器**

在此部分中，需要在 csim.c 文件里编写一个缓存模拟器，该模拟器以 valgrind 内存跟踪作为输入，模拟此跟踪上的高速缓存存储器的命中和未命中行为，并输出命中、未命中和排出的总数目。

实验提供了参考缓存模拟器的二进制可执行文件 csim-ref，在 valgrind 跟踪文件上模拟具有任意大小和关联性的高速缓存的行为。它使用选择 LRU 替换策略。参考模拟器采用的命令行参数如下：

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

如何处理这 6 个参数是需要解决的第一个问题。由于 argc 保留

了输入参数的个数，argv 保存了输入的参数。所以可以直接利用 main 函数的参数获得我们需要的所有参数。代码如下：

```
char *path;
int char2int(char c){
	return c-'0';
}
int main(int argc,char* argv[])
{
	int s,E,b,S;
	FILE *fp;
	for(int i=1;i<argc;i++) {
		if (argv[i][0]=='-'){
			char tag=argv[i][1];
			switch (tag) {
				case 's':
					s=char2int(argv[++i][0]);
					break;
				case 'E':
					E=char2int(argv[++i][0]);
					break;
				case 'b':
					b=char2int(argv[++i][0]);
					break;
				case   't':
					path=argv[++i];
					break;
				default:
					break;
			}
		}
		if(i>argc)break;
	}
}
```

缓存模拟器需要包括 tag、valid 和 stamp，这里的 stamp 是为了 LRU 算法做准备。由于一组可以有多行，所以可以将缓存定义为一个二维的结构体数组。代码如下：

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

```
    int stamp;
}CacheLine;
CacheLine **Cache= NULL;
void InitCache(int S,int E){
    Cache = ( CacheLine**)malloc(sizeof(CacheLine*) * S);
    for(int i=0;i<E;i++){
        Cache[i]=( CacheLine*)malloc(sizeof( CacheLine)*E);
        for(int j=0;j<E;j++){
            Cache[i][j].tag=-1;
            Cache[i][j].stamp=-1;
            Cache[i][j].valid=0;
        }
    }
}
void FreeCache(int S){
    for(int i=0;i<S;i++)free(Cache[i]);
    free(Cache);
}
```

由于缓存模拟器不需要管 Load 和 Store 之后的操作，而 modify 操作则是一次读一次写，既调用两次 update 函数，具体步骤如下：

1. 首先解析出地址中的 s 和 tag；
2. 确定组之后去对应的组里遍历行，如果有相同的 tag 位则命中；
3. tag 位不相同则表示未命中，此时如果有空闲位置则直接写入；
4. 没有空闲位置，需要换出 miss++及 evits++。

```
void update(uint64_t address,int b,int s,int E){
    int index = (address >> b) & ((-1U) >> (64 - s));
    int tag = address >> (b + s);
    int MaxStamp=INT_MIN;
    int MaxStamp_index=-1;
    for(int i=0;i<E;i++){
        if(Cache[index][i].tag==tag)
        {
            Cache[index][i].stamp=0;
            ++Hits;
            return ;
        }
    }
    for(int i=0;i<E;i++){
```

```
            if (Cache[index][i].valid==0){
                Cache[index][i].stamp=0;
                Cache[index][i].valid=1;
                Cache[index][i].tag=tag;
                ++Misses;
                return ;
            }
        }
        ++Evicts;++Misses;
        for(int i = 0; i < E; ++i)
        {
            if(Cache[index][i].stamp > MaxStamp)
            {
                MaxStamp = Cache[index][i].stamp;
                MaxStamp_index = i;
            }
        }
        Cache[index][MaxStamp_index].tag = tag;
        Cache[index][MaxStamp_index].stamp = 0;
        return ;
}
```

最后释放所有的动态申请并关闭对应打开的文件，完整代码查看 csim.c 文件。运行结果如下：

```
                        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)       4       5       2       4       5       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
```

**实验 B：优化矩阵转置**

本实验要求修改 trans.c 文件中的 transpose_submit 函数，使矩阵转置时的不命中次数尽可能小，并分别对 32*32、64*64 和 61*67 的矩阵进行实验。实验要求如下：

1. 最多只能定义 12 个局部变量；

2. 不使用数组，不调用任何类似于 malloc 的开辟内存函数；

3. 不能修改原始的矩阵；

4. 不能使用递归函数。

**32*32 矩阵**

采用分块技术，因为 cache 一行能放 8 个，分块也用 8 的倍数。在 32*32 矩阵中，一行有 32 个 int，即 4 个 cache line，所以可以采用长宽为 8 的分块。此外，除了循环需要的 4 个变量外我们还剩余 8 个自由变量可用，正好可以存一个 cache line。一次性读完一行，减少未命中数，代码如下：

```
if(M == 32)
{
        int i, j, k, v1, v2, v3, v4, v5, v6, v7, v8;
        for (i = 0; i < 32; i += 8)
            for(j = 0; j < 32; j += 8)
                for(k = i; k < (i + 8); ++k)
                {
                    v1 = A[k][j];
                    v2 = A[k][j+1];
                    v3 = A[k][j+2];
                    v4 = A[k][j+3];
                    v5 = A[k][j+4];
                    v6 = A[k][j+5];
                    v7 = A[k][j+6];
                    v8 = A[k][j+7];
                    B[j][k] = v1;
                    B[j+1][k] = v2;
                    B[j+2][k] = v3;
                    B[j+3][k] = v4;
                    B[j+4][k] = v5;
                    B[j+5][k] = v6;
                    B[j+6][k] = v7;
                    B[j+7][k] = v8;
                }
}
```

**64*64 矩阵**

对 64*64 的矩阵而言，每行有 64 个 int，如果仍然使用 8*8 的

分块，会因为映射到了相同的块而造成冲突，所以采用对 8 分块再进行 4 分块的方法，代码如下：

```
else if (M == 64)
{
        int i, j, x, y;
        int x1, x2, x3, x4, x5, x6, x7, x8;
        for (i = 0; i < N; i += 8)
            for (j = 0; j < M; j += 8)
            {
                for (x = i; x < i + 4; ++x)
                {
                    x1 = A[x][j]; x2 = A[x][j+1]; x3 = A[x][j+2]; x4 = A[x][j+3];
                    x5 = A[x][j+4]; x6 = A[x][j+5]; x7 = A[x][j+6]; x8 = A[x][j+7];
                    B[j][x] = x1; B[j+1][x] = x2; B[j+2][x] = x3; B[j+3][x] = x4;
                    B[j][x+4] = x5; B[j+1][x+4] = x6; B[j+2][x+4] = x7; B[j+3][x+4]
= x8;
                }
                for (y = j; y < j + 4; ++y)
                {
                    x1 = A[i+4][y]; x2 = A[i+5][y]; x3 = A[i+6][y]; x4 = A[i+7][y];
                    x5 = B[y][i+4]; x6 = B[y][i+5]; x7 = B[y][i+6]; x8 = B[y][i+7];
                    B[y][i+4] = x1; B[y][i+5] = x2; B[y][i+6] = x3; B[y][i+7] = x4;
                    B[y+4][i] = x5; B[y+4][i+1] = x6; B[y+4][i+2] = x7; B[y+4][i+3]
= x8;
                }
                for (x = i + 4; x < i + 8; ++x)
                {
                    x1 = A[x][j+4]; x2 = A[x][j+5]; x3 = A[x][j+6]; x4 = A[x][j+7];
                    B[j+4][x] = x1; B[j+5][x] = x2; B[j+6][x] = x3; B[j+7][x] = x4;
                }
            }
}
```

**61*67 矩阵**

由于不是长宽相等的矩形，所以不一定使用 8 分块，采用最简单的方法，一个一个尝试不同的分块大小，17 分块可以达到最少的未命中数。我们可以对对角线用 8 分块做一些操作，代码如下：

```
else if(M == 61)
{
```

```
    int i, j, v1, v2, v3, v4, v5, v6, v7, v8;
    int n = N / 8 * 8;
    int m = M / 8 * 8;
    for (j = 0; j < m; j += 8)
        for (i = 0; i < n; ++i)
        {
            v1 = A[i][j];
            v2 = A[i][j+1];
            v3 = A[i][j+2];
            v4 = A[i][j+3];
            v5 = A[i][j+4];
            v6 = A[i][j+5];
            v7 = A[i][j+6];
            v8 = A[i][j+7];
            B[j][i] = v1;
            B[j+1][i] = v2;
            B[j+2][i] = v3;
            B[j+3][i] = v4;
            B[j+4][i] = v5;
            B[j+5][i] = v6;
            B[j+6][i] = v7;
            B[j+7][i] = v8;
        }
    for (i = n; i < N; ++i)
        for (j = m; j < M; ++j)
        {
            v1 = A[i][j];
            B[j][i] = v1;
        }
    for (i = 0; i < N; ++i)
        for (j = m; j < M; ++j)
        {
            v1 = A[i][j];
            B[j][i] = v1;
        }
    for (i = n; i < N; ++i)
        for (j = 0; j < M; ++j)
        {
            v1 = A[i][j];
            B[j][i] = v1;
        }
}
```

最终运行结果如下：

```
Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1219
Trans perf 61x67          10.0        10        1813
```