华中科技大学

计算机系统结构实验报告

姓　　名：	徐锦慧
学　　院：	计算机科学与技术
专　　业：	计算机科学与技术
班　　级：	CS2011
学　　号：	U202011675
指导教师：	施展

分数	
教师签名	

2023 年 05 月 06 日

目　录

1 Cache 模拟器设计

1.1 实验目的

1. 实验目的

1) 深入了解计算机系统中高速缓存的工作原理，包括缓存的层次结构、缓存行、缓存映射等概念，并能够实现一个模拟的高速缓存。
2) 了解缓存的性能瓶颈，并能够使用性能分析工具来确定性能问题，并提出解决方案。
3) 掌握一些调试技术，例如使用 GDB 调试器和 Valgrind 内存调试器。

2. 实验任务

- 输入：内存访问轨迹（src\traces 子文件夹中的*.trace 文件）。
- 操作：模拟缓存相对内存访问轨迹的命中（hit）/缺失行为（miss）。
- 输出：基于 LRU 算法的命中、缺失和（缓存行）脱胎/驱除的总数。

总的来说，要求在 csim.c 文件中实现一个 Cache 模拟器，使其能够模拟指定大小和关联度的高速缓存的运作过程。

3. 实验要求

1) 模拟器必须在输入参数 s,E,b 设置为任意值时均能正确工作,即需要使用 malloc 函数来为模拟器中的数据结构分配存储空间。
2) 本实验仅关心数据 Cache 的性能，因此模拟器应忽略所有指令 cache 访问，即不考虑轨迹中“I”起始的行。
3) 假设内存访问的地址总是正确对齐的，即一次内存访问从不跨越块的边界，可忽略访问轨迹中给出的访问请求大小。
4) 必须在 main 函数最后调用 printSummary 函数，并如下传之以命中 hit、缺失 miss 和淘汰/驱逐 eviction 的总数作为参数：

1.2 实验环境

【主机机带】RAM 8.00 GB (7.75 GB 可用)

【处理器】Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz

【主机版本】Windows 10 专业版

【虚拟机版本】 Ubuntu 20.04.4 LTS 64-bit

【实验平台】Visual Studio Code 1.63.0

1.3 实验思路

1.3.1 分析给定文件

首先观察 src/traces 中的测试集文件，如图 1.1 所示。其中每一行代表一次对缓存的操作，格式为： [空格] 操作 地址,数据大小。

操作的类型有 I、L、S、M 这 4 种，只有“I”操作前无空格。其中“I”表示指令加载；“L”表示数据加载；“S”表示数据存储；“M”表示数据修改，即数据存储之后的数据加载。

地址指定一个 32 位的十六进制存储器，数据大小则是指操作访问的字节数。

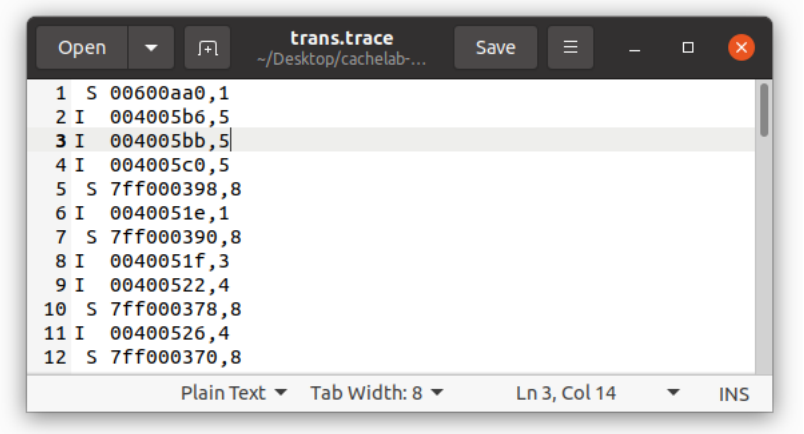

图 1.1 trace 文件内容

1.3.2 数据结构及变量定义

实验要求实现的缓存存储器的行为和 csim-ref 一致，使用 LRU 算法进行替换操作。根据参考教材，高速缓存存储器可以用四元组 (S,E,B,m) 来描述，其中 $S = 2^s$ 为组数，E 为行数，$B = 2^b$ 为块的大小，m 为地址的位数，具体结构如图 1.2 所示。

我们定义一个数据结构 CacheLine，用来表示高速缓存中的行。其中，valid 表示有效位，tag 表示标记，同时题目要求使用 LRU 替换算法，因此我们用 time 表示与上次访问的间隔。由于题目没有要求存储数据，结构中并未包含缓存块的组数。具体声明代码如下：

```
typedef struct{
    int valid;  // 有效位
    int tag;    // 标签
    int time;   // 时间戳，记录上一次访问这条缓存的时间
```

```
} CacheLine, *CacheSet, **Cache;
```

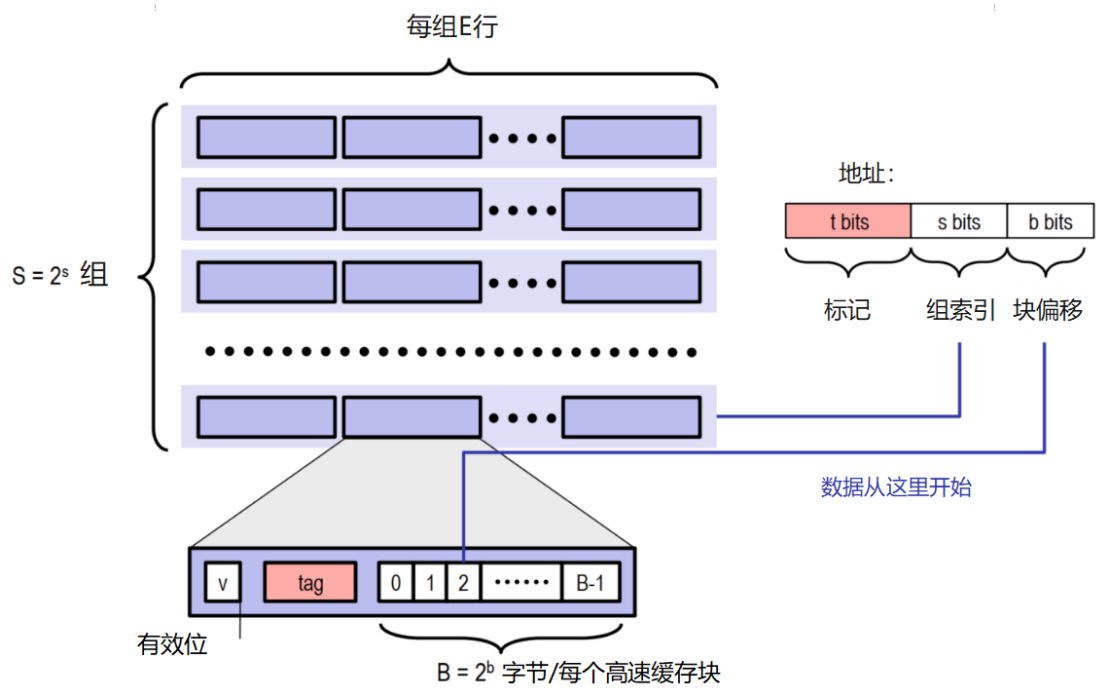

图 1.2 cache 结构

其他全局变量定义如下：

```
int hit, miss, evict;  // 表示访问结果
int s, S, E, b;          // 缓存接收的个参数：组索引位数、组数、行数、块大小的 b
char filePath[100];    // trace 文件的路径
Cache cache;           // 整个缓存
```

hit，miss，evict 分别记录命中、缺失、淘汰 3 种访问结果的出现次数。s 表示组索引的位数；S 表示高速缓存索引的组数；E 表示每组索引的行数；b 表示块偏移的大小。filePath 是 trace 文件的路径，而 cache 代表整个高速缓存。

对于模拟的高速缓存索引，至少需要接收 4 个参数：s、E、b、t。

1.3.3 代码框架

我们在 main() 函数中完成命令行参数解析和模拟工作，具体代码如下：

```
int main(int argc, char* argv[])
{
    int opt;
    // getopt 在循环中调用来检索参数，它的返回值存储在一个局部变量中
    while ((opt = getopt(argc, argv, "s:E:b:t:")) != -1) {
        switch (opt) {
            case 's':
                s = atoi(optarg);
```

```
                S = 1 << s;break;
            case 'E':
                E = atoi(optarg);break;
            case 'b':
                b = atoi(optarg);break;
            case 't':
                strcpy(filePath, optarg);break;
        }
    }

    mallocCache();  // 动态分配缓存空间
    simulate();     // 模拟缓存的读写操作
    freeCache();    // 释放缓存空间
    printSummary(hit, miss, evict);  // 输出结果
    return 0;
}
```

首先，我们使用 getopt() 函数来进行命令行分析。在 while 循环中调用它来检索参数，其返回值存储在变量 opt 中。当函数返回-1 时，表示没有操作，否则判断 opt 是 s、E、b、t 中的哪一个，并将其转换为对应类型。

由于 s、E 和 b 是变量，所以需要使用 malloc() 函数来在堆上动态分配空间，这部分内容在 mallocCache() 中统一实现。

接着我们在 simulate 中模拟缓存的读写操作，结束后使用 freeCache() 及时释放之前申请的空间，并使用 printSummary() 函数输出结果。

1.3.4 函数描述

1. mallocCache()

函数功能：在堆上为 cache 动态分配空间。

实现思路：首先用 malloc() 函数给整个 Cache 分配空间，大小和 S 组 CacheSet 的相同。接着给每组 CacheSet 分配空间，大小和 E 行 CacheLine 的相同，并将其初始化为 0。代码如下：

```
void mallocCache(){
    // 先给整个 cache 分配空间
    cache = (Cache)malloc(S * sizeof(CacheSet));
    assert(cache);

    // 再给每一行的 CacheLine 分配空间并初始化
    for(int i=0; i<S; i++){
```

```
        cache[i] = (CacheSet)malloc(E * sizeof(CacheLine));
        assert(cache[i]);
        memset(cache[i],0,sizeof(CacheLine)*E);
    }
}
```

2. freeCache()

函数功能：释放之前申请的缓存空间。

实现思路：和 mallocCache()的顺序相反，首先用 free()释放每个 cacheSet 的空间，再释放整个 Cache。代码如下：

```
void freeCache(){
    // 先释放每一个 CacheLine
    for(int i=0; i<S; i++)
        free(cache[i]);

    // 再释放整个 cache
    free(cache);
}
```

3. simulate()

函数功能：根据 trace 文件模拟缓存的读写操作。

实现思路：该函数流程图如图 1.3 所示。

1) 根据 filePath 打开测试文件
2) 通过 fscanf() 函数从文件中一行行读出 3 个参数：操作 opt、存储器地址 address、操作访问的字节数 size。
3) 根据 opt 的值模拟对应的访存操作。如果 opt 为'L'或'S'，只需访问缓存一次；如果 opt 为'M'，需要访问两次。
4) 执行对应的操作后，更新时间戳。
5) 若文件中还有待读取的行，则返回第 2) 步继续执行，否则关闭文件。

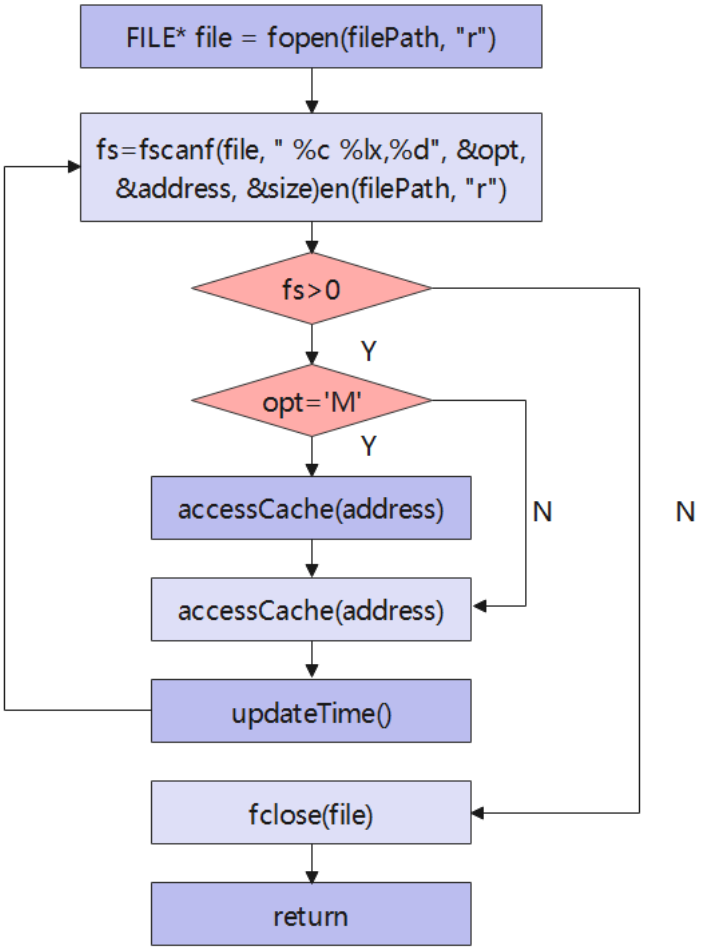

图 1.3 simulate()流程图

4. accessCache(uint64_t address)

函数功能：模拟访问缓存的过程。输入存储器地址 address，对 cache、cacheSet 变量进行相应操作，并根据其访问结果更新 hit、miss、evict 的值。

实现思路：该函数流程图如图 1.4 所示。

1) 根据输入的 address 得到标记 tag 和组索引，再通过组索引得到组 cacheSet。
2) 遍历 cacheSet 进行行匹配。如果当前元素有效位为 1 且 tag 与地址中的标记相同，则说明缓冲击中，更新 hit 值和该元素的时间戳并退出；否则 miss 值加一，继续执行。
3) 缓冲未击中，再次遍历 cacheSet 看是否有空位。如果找到空位，则将数据写入空行，即更新该元素的有效值、标记和时间戳，更新后退出；否则 evict 值加一，继续执行。

4) 缓冲未击中并且无空位，需要用 LRU 算法进行替换。首先遍历 cacheSet，找到时间戳最大的元素下标 idx，此即最久未访问的元素，再更新其 tag 值和时间戳，实现替换操作。

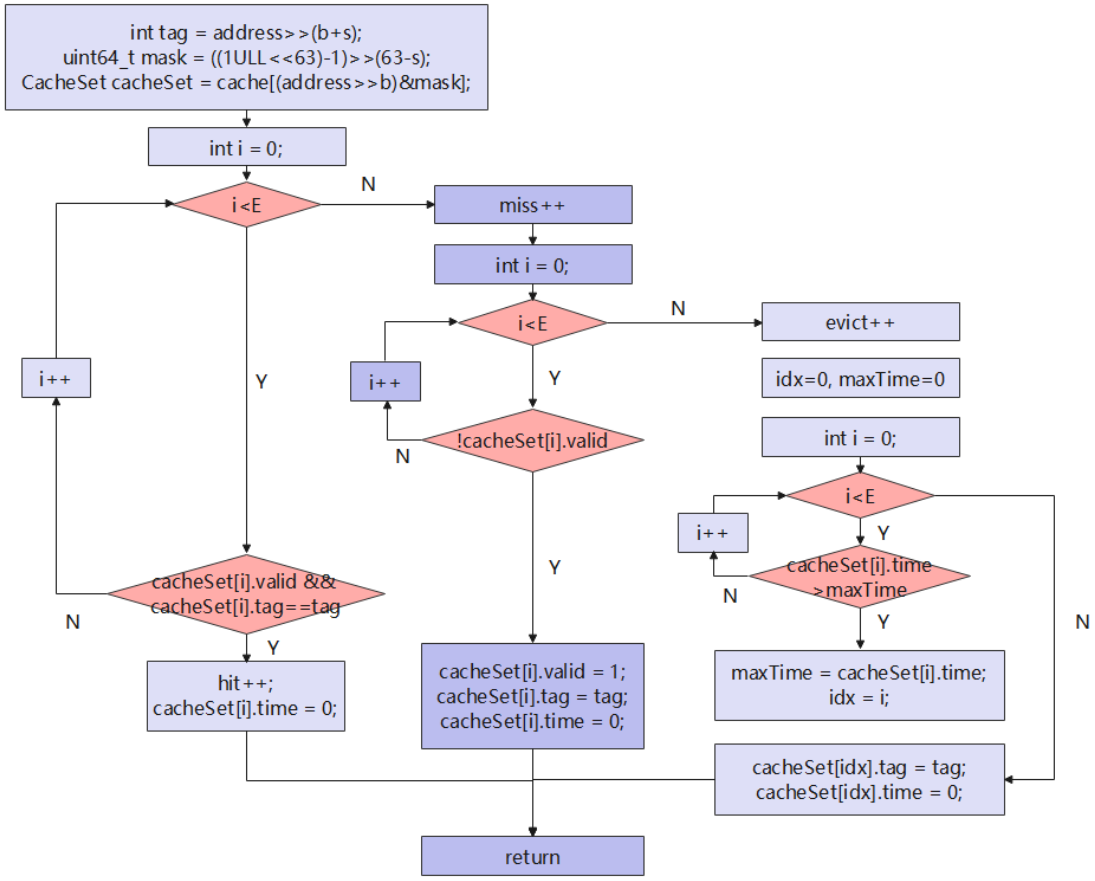

图 1.4 accessCache()流程图

5. updateTime()

函数功能：在每次访问缓存操作后更新时间戳。

实现思路：遍历 cache 的每一组每一行元素，如果有效位为 1 则将其时间戳加一。

```
void updateTime(){
    for(int i=0; i<S; i++){
        for(int j=0; j<E; j++){
            if(cache[i][j].valid)
                cache[i][j].time++;
        }
    }
}
```

1.4 实验结果和分析

最初按照以上分析实现对应函数，得到如图 1.5 所示的结果：

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     2 (1,1,1)       9       8       8       9       8       6  traces/yi2.trace
     2 (4,2,4)       4       5       5       4       5       2  traces/yi.trace
     2 (2,1,4)       2       3       3       2       3       1  traces/dave.trace
     2 (2,1,3)     167      71      71     167      71      67  traces/trans.trace
     2 (2,2,3)     201      37      37     201      37      29  traces/trans.trace
     2 (2,4,3)     212      26      26     212      26      10  traces/trans.trace
     2 (5,1,5)     231       7       7     231       7       0  traces/trans.trace
     4 (5,1,5)  265189   21775   21775  265189   21775   21743  traces/long.trace
    18

TEST_CSIM_RESULTS=18
```

图 1.5 错误结果

由图可知，Hits 和 Misses 的结果与给出的标准答案一致，但 Evicts 的数值有误。经过检查发现，是 accessCache() 函数出错，在缓存未击中且写入空位后没有及时返回，导致 Evicts 数值增多。最终修改完得到正确结果，如图 1.6 所示：

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./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)       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

TEST_CSIM_RESULTS=27
```

图 1.6 正确结果

2 优化矩阵转置

2.1 实验目的

1. 实验目的

1) 掌握如何使用编译器来生成高效的汇编代码，以减少缓存未命中和提高程序性能。
2) 掌握使用 loop unrolling、loop fusion 和 blocking 等技术来优化程序性能的基本方法。
3) 深入了解计算机系统中高速缓存的工作原理。

2. 实验任务

在 trans.c 中使用 C 语言编写一个实现矩阵转置的函数 transpose_submit()。即对于给定的矩阵 Am×n，得到矩阵 Bn×m，使得对于任意 0<=i<n、0<=j<m,有 B[j][i]=A[i][j]，其并且使函数调用过程中对 cache 的不命中数 miss 尽可能少。

3. 实验要求

1) 限制对栈的引用。在转置函数中最多定义和使用 12 个 int 类型的局部变量，同时不能使用任何 long 类型的变量或其他位模式数据以在一个变量中存储多个值。
2) 不允许使用递归。
3) 转置函数不允许改变矩阵 A，但可以任意操作矩阵 B。
4) 不允许在代码中定义任何矩阵或使用 malloc 及其变种。

4. 评判标准

针对每一矩阵大小，性能分数线性依赖于发生的 Cache 缺失总数 m。

- 32×32：如果 m<300 得 10 分，如果 m>600 得 0 分，对其他 m 得(600-m)*10/300 分。
- 64×64：如果 m<1300 得 10 分，如果 m>2000 得 0 分，对其他 m 得(2000-m)*10/700 分。
- 61×67：如果 m<2000 得 20 分，如果 m>3000 得 0 分，对其他 m 得(3000-m)*20/1000 分。

2.2 实验环境

【主机机带】RAM 8.00 GB (7.75 GB 可用)

【处理器】Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz

【主机版本】Windows 10 专业版

【虚拟机版本】 Ubuntu 20.04.4 LTS 64-bit

【实验平台】Visual Studio Code 1.63.0

2.3 实验思路

2.3.1 分析给定文件

首先观察给定应的自动测试文件 test-trans.c，发现该程序调用了 trans.c 中的 registerFunctions() 函数，使用 valgrind 追踪转置函数并生成轨迹文件，再调用实验 1 中的 Cache 模拟器就能统计出转置函数的 miss 数。我们需要完成 trans.c 文件中的 transpose_submit() 函数来实现矩阵的转置，同时要使 cache miss 的次数尽可能少。

实验已经给定了 cache 的参数：s = 5，b = 5，E = 1。因此可以得出 cache 的大小就是 32 组，每组 1 行，每行可存储 32 字节的数据，即 8 个 int 型数据。

我们以 4*4 的数组 A、B 为例，A 数据组按行访问，B 数组按列访问，矩阵转置的效果如图 2.1 所示。

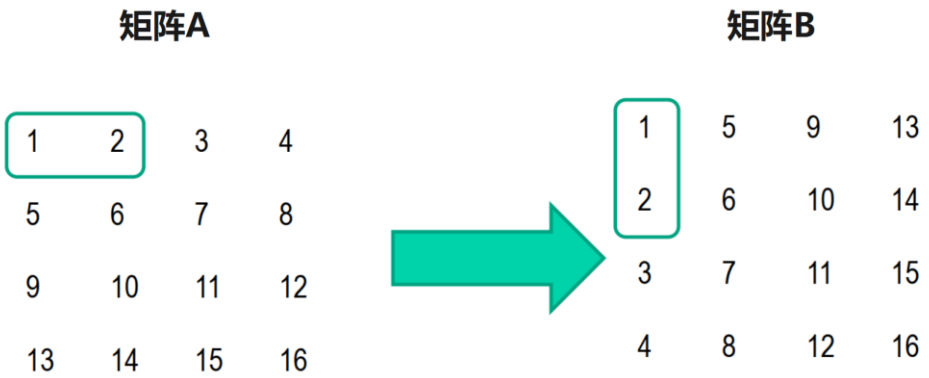

图 2.1 矩阵转置效果

A 数组访问 A[0][0]，不命中，将块 1 装入 cache。B 数组访问 B[0][0]，虽然 B[0][0]所映射的块 1 在 cache 中，但是标记位不同，造成冲突不命中，重新将数组 B 对应的块 1 装入 cache。

A 数组访问 A[0][1]，虽然 A[0][1] 所映射的块 1 在 cache 中，但是标记位不同，造成冲突不命中，重新将数组 A 对应的块 2 装入 cache。B 数组访问 B[1][0]，虽然 B[1][0]所映射的块 2 在 cache 中，但是标记位不同，造成冲突不命中，重新

将数组 B 对应的块 2 装入 cache。

分析上述可以看出 miss 过多的原因在于访问两个数组的过程中存在太多冲突，而造成 miss 原因是 B 数组和 A 数组中下标相同的元素会映射到同一个 Cache 块，但标记位置不同。

因此，要减少 miss 数，就要解决冲突不命中的问题。基于此，我们可以一次性访问同一个块中的多个元素，访问完以后便不再需要访问这个块了，从而可以大大地减少冲突不命中的数目。故完成该实验的重要方法就是分块。

2.3.2 32×32 矩阵

该测试矩阵大小为 32 x 32。我们先使用原始的示例代码测试，发现 miss 数为 1183 个，而满分的要求为 300 个，因此我们需要使用分块技术优化。

又因为 cache 一行能放 8 个 int 型数据，所以我们分块最好也用 8 的倍数。在 32x32 的矩阵中，一行有 32 个整数，需要 4 个 cache 行，所以 cache 一共可以存矩阵的 8 行，正好可以用长宽都为 8 的分块。于是我们按 8x8 的分块改写代码并调试。

测试结果表明，采用 8x8 分块后，miss 数降低为 343，但仍未达到 300 的要求。A 和 B 中相同位置的元素是映射在同一 cache line 上的，但是因为我们转置时，并不会把 A 中某位置的元素放到 B 中相同的地方，除非在对角线上，此时就会发生原地转置。譬如我们已经把 A 的第四行存进去了，但是当要写 B[4][4] 时，发生了冲突，第四行被换成 B 的，然后读 A 的时候又换成了 A 的，就多造成了两次 miss。所以，我们应该避免对角线上元素原地转置引发的冲突未命中问题，使用循环展开直接访问行中的 8 个元素并赋值给 B。核心代码如下：

```
    if(M==32 && N==32){
        for (i = 0; i < 32; i+=8){
            for (j = 0; j < 32; j+=8){
                for(k = i; k < i+8; k++){
                        t0 = A[k][j];
                        t1 = A[k][j+1];
                        t2 = A[k][j+2];
                        t3 = A[k][j+3];
                        t4 = A[k][j+4];
                        t5 = A[k][j+5];
                        t6 = A[k][j+6];
```

```
                    t7 = A[k][j+7];

                    B[j][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;
                }
            }
        }
    }
```

再次测试，miss 降低为 287，符合要求。

2.3.3 64×64 矩阵

对于 64 x 64 的矩阵来说，每行有 64 个 int 型数据，则 cache 只能存储矩阵的 4 行了，所以如果使用 8x8 的分块，一定会在写 B 的时候造成冲突，因为映射到了相同的块。用 8×8 的分块方法测试，发现 miss 结果为 4611，效果并不理想。

接着尝试原始的 4×4 分块，发现 miss 数降低到了 1699，但还是会浪费一次读加载 8 个的特性，和要求的 1300 相比仍有差距。

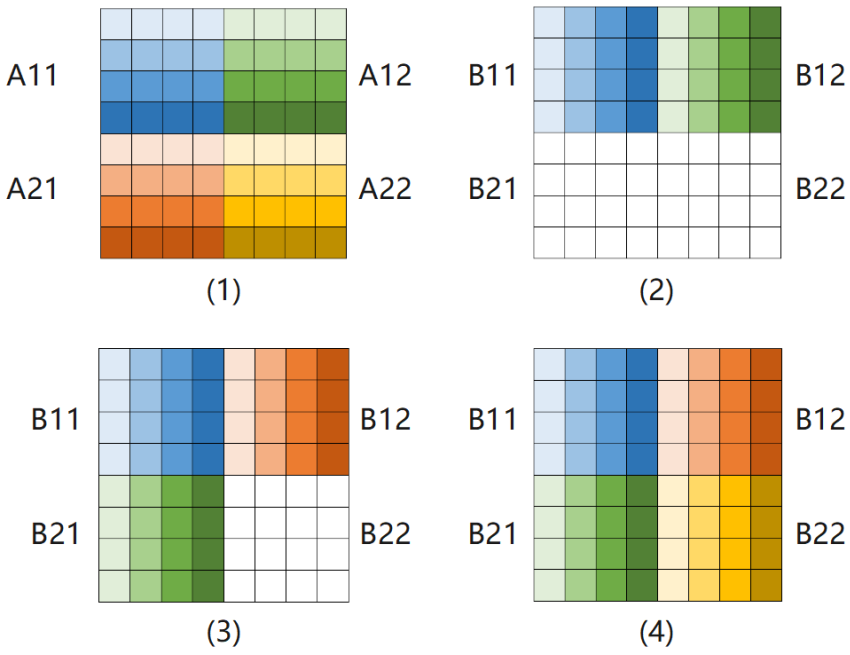

图 2.2 分块思路

于是转换思路，将 8×8 的分块再分为 4 个 4×4 的小块处理，同时利用块间已经加载的缓存的特性。根据提示“不可以更改矩阵 A 的值，但是可以更改矩阵 B”，可以利用 B 已经加载进来的部分存数据而不引起新的 miss。利用这一思想继续优化程序。写数据时如果已经在缓存里了称为 warm，否则称为为 cold，下面使用 cold rate 指标来分析性能。思路如图 2.2 所示。

1) 将 8×8 的分块再分为 4 个 4×4 的小块处理，A 的 cold rate 为 1/2。
2) 读入 A11、A12，并且把 A12 转置存进 B12 而非 B21 中，避免造成额外的 cold rate，此时 B 的 cold rate=1/4。
3) 把 B12 读出来平移到 B21，再读 A21 并把它转置到 B12。这样操作虽然造成了 B 的最后一个 cold rate，但接下来再写 B22 时就不必再加载了。
4) 将 A22 转置到 B22，此时 B 的 cold rate 只有 2/4，性能就达到题目要求。测试结果表明，miss 值降低为 1179。

2.3.4 61×67 矩阵

该矩阵为不规则的矩阵，本质也是用分块来优化 Cache 的读写，但是不能找到比较显然的规律看出来间隔多少可以填满一个 Cache。但是由于要求比较松，依次尝试一些分块的大小，直接进行转置操作。经过测试发现，当分块大小为 16×16 时，结果为 1992 次 miss，满足要求。具体代码如下：

```
if(M==61 && N==67){
    for(i = 0; i < 67; i += 16){
        for(j = 0; j < 61; j += 16){
            for(t0 = i; t0<i+16 && t0<67; t0++){
                for(t1 = j; t1<j+16 && t1<61; t1++){
                    B[t1][t0] = A[t0][t1];
                }
            }
        }
    }
}
```

2.4 实验结果和分析

2.4.1 32×32 矩阵

首先使用原始的示例代码测试，得到 miss 数为 1183，如图 2.3 所示。

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./test-trans -M 32 -N 32

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:870, misses:1183, evictions:1151

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:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=1183

TEST_TRANS_RESULTS=1:1183
```

图 2. 3 miss 数为 1183

接着按 8x8 的分块改写代码来直接转置，得到 miss 数为 343，如图 2.4 所示。

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./test-trans -M 32 -N 32

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:1710, misses:343, evictions:311

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:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=343
```

图 2. 4 miss 数为 343

对 8x8 分块改进，解决对角线上元素原地转置引发的冲突未命中问题后，miss 数降低为 287，满足 miss<300 的要求，如图 2.5 所示。

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./test-trans -M 32 -N 32

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:1766, misses:287, evictions:255

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:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

图 2. 5 miss 数为 287

2.4.2 64×64 矩阵

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./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:6498, misses:1699, evictions:1667

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=1699

TEST_TRANS_RESULTS=1:1699
```

图 2.6 miss 数为 1699

直接采用 4×4 分块，miss 数为 1699，如图 2.6 所示。

接着将 8×8 的分块再分为 4 个 4×4 的小块处理，同时利用块间已经加载的缓存的特性，miss 降低为 1179，满足 miss<1300 的要求，结果如图 2.7 所示。

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./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:9066, misses:1179, evictions:1147

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=1179

TEST_TRANS_RESULTS=1:1179
```

图 2.7 miss 数为 1179

2.4.3 61×67 矩阵

通过采用 16×16 分块对矩阵直接转置，miss 数降低为 1992，满足 miss<2000 的要求，结果如图 2.8 所示。

```
xjh@ubuntu:~/Desktop/cachelab-handout$ ./test-trans -M 61 -N 67

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:6187, misses:1992, evictions:1960

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:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1992

TEST_TRANS_RESULTS=1:1992
```

图 2.8 miss 数为 1992

3 总结和体会

3.1 Cache 模拟器设计

本次Cache模拟器设计实验的主要目的是加深对计算机系统中高速缓存的工作原理的理解，包括缓存的层次结构、缓存行、缓存映射等概念。

通过完成此次实验，我对计算机系统底层的缓存机制有了更深入的了解。在实现 Cache 模拟器时，我不仅学习了高速缓存的基本原理和缓存的读写操作，还深入了解了缓存的行、组和块等概念，以及如何通过缓存的映射和替换策略来提高缓存的命中率。

在实验刚开始时，对实验的要求和所给资料比较迷茫，通过和同学交流、查阅相关资料成功解决了自己的疑惑，同时也提高了自己解决问题、搜索信息的能力。

实验过程中也遇到了一些bug，通过分析代码、反复调试，最终也得到了正确结果。我对GDB调试器和Valgrind内存调试器也有了更深入的了解，掌握了一些调试的技巧。

3.2 优化矩阵转置

矩阵转置实验的主要目的是了解计算机系统中高速缓存的工作原理，能够通过分块等技术来优化一个矩阵转置程序，使miss数达到指定要求。

实验分为3个部分，其中61×67的矩阵最为简单，可以直接通过尝试不同的分块得出；32×32 的矩阵则需要考虑对角线上元素的冲突未命中问题；64×64的矩阵最难，需要多次尝试。

通过此次实验，我学习了如何使用不同的分块方法来减少miss数，从而提高程序的性能。同时，我还学习了如何使用性能分析工具来检查程序的性能瓶颈，并使用编译器的优化参数来进一步提高程序效率。此外，我也深入了解了编译器的工作原理，并学习到了如何编写高效的代码。

总的来说，Cache Lab 实验是非常有意义的一个实验。通过完成这个实验，我更深入了解了计算机系统底层和编译器的工作原理，并学习到了如何编写高效的代码和进行性能优化。这些知识将对我的计算机科学学习和职业发展都会产生深远的影响。

4 对实验课程的建议

本实验源自 CSAPP，主要任务是编写 Cache 模拟器和优化矩阵乘法程序。作为一项计算机系统底层实验，Cache Lab 可以非常好地帮助我们深入理解计算机系统底层的缓存机制，其设计非常实用且有意义。

此外，本实验的指导和要求非常清晰明了，由于知名度较高，可以参考的资源也很多。实验环境也十分稳定和可靠，保证了实验过程的顺利进行。在完成本实验的过程中，需要不断调整和优化程序，从而提高程序的效率和性能，增强了我们的实验能力和创新能力。

Cache 模拟器的编写难度适中，实验设计也很有趣，但优化矩阵乘法略显困难，尤其是在完成 64×64 矩阵时。希望老师可以在这部分给予更多的讲解，着重介绍一下分块技术，以此获得更好的实验体验。

同时也也可以提供更多的教学资源，例如实验讲解视频、实验辅导指南等，可以帮助学生们能够更好地理解和掌握实验要求。

参考文献

[1] 张晨曦，张晨曦，王志英, 等. 计算机系统结构教程[M]. 清华大学出版社, 2009.

[2] RandalE.Bryant, DavidO'Hallaron，龚奕利, 等. 深入理解计算机系统[J]. 中国电力出版社，2004.

[3] https://zhuanlan.zhihu.com/p/79058089

[4] http://csapp.cs.cmu.edu/3e/cachelab.pdf