华中科技大学

计算机系统结构实验报告

姓　　名：

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：

学　　号：

分数	
教师签名	

2023 年.　 4 月.　28 日

目　录

1. Cache 模拟器实验

1.1. 实验目的

本 lab 一共分为两个部分。第一部分为在给定框架下完成缓存模拟器的编写。第二部分为优化矩阵转置函数，使其缓存 miss 数达到要求。

1.2. 实验环境

Ubuntu 22.04
Python 2.7.18
gcc 11.0.3

1.3. 实验思路

1.3.1 缓存模拟器

第一部分为缓存模拟器的编写。可以看到给出的框架下缺少对 cache 行的数据结构定义，因此首先对此进行补足。由实验要求知，采用的是 LRU 替换算法，因此结构体中需要记录该块未被访问的时间，定义为 time。其余部分即为 valid 位与 tag 位。

```
/*line struct*/
typedef struct {
   int tag;
   bool valid;
   int time; //LRU timer
} line_t;
```

随后，将 line 进行编组，也采用结构体的方式。

```
/*set struct*/
typedef struct {
   line_t *lines;
} set_t;
```

接下来是缓存的定义。缓存由许多组组成，属性包括组数与每组的行数。

```
/*cache struct*/
typedef struct {
```

```
    set_t *sets;
    size_t set_num;
    size_t line_num; // number of lines per set
} cache_t;
```

cache 的相关定义完成后，开始进行 cache 的初始化。

```
/*cache initialization*/
cache_t cache = {};
int set_index_bits = 0; //index
int block_bits = 0; //offset
```

由于本关最后需要调用 printSummary(hit_count, miss_count, eviction_count)，因此在进入主函数前对 hit_count，miss_count，eviction_count 三个变量进行初始化。

在任务要求中提到，高速缓存器的结构可以用元组(S,E,B,m)来描述，在命令行中也包括了这几个参数。实验还提供了可以读取 unix 命令行的函数 getopt()。该函数读取一个命令行，并设置*optstring 字符串以指定选项字符，若某选项字符后有冒号（:），则代表其需要参数。

以下为命令行解析的实现，原理非常简单，即在读取到对应选项字符时，对后续传入的参数进行读取处理。在读取完成后，需要进行异常检测，观察每一条命令执行是否成功。

```
for (int opt; (opt = getopt(argc, argv, "s:E:b:t:")) != -1;)
{
    switch (opt) {
      /*2)compute s, E, b from command line arguments*/
      case 's':
        set_index_bits = atoi(optarg);
        cache.set_num = 2 << set_index_bits;
        break;
      case 'E': //number of line per set
        cache.line_num = atoi(optarg);
        break;
      case 'b':
        block_bits = atoi(optarg);
        break;
      case 't': // Input filename
        if (!(file = fopen(optarg, "r"))) {
```

```
            return 1;
        }
            break;
          default://otherwise, it is an unknown option
            return 1;
        }
    }
    if ((!set_index_bits) || (!cache.line_num) || (!block_bits) || (!file)) {
      return 1;
    }
```

命令行解析完成后，便是对 cache 进行初始化。首先对 cache.sets 进行空间分配，分配完成后，对 set 内的 lines 进行空间分配。

```
    cache.sets = malloc(sizeof(set_t) * cache.set_num);
    for (int i = 0; i < cache.set_num; i++) {
      cache.sets[i].lines = calloc(sizeof(line_t), cache.line_num);
    }
```

访存指令的格式在要求中已经给出。operation 中，“I”表示指令加载，(由于本实验仅关心数据 Cache 的性能，因此模拟器应忽略所有指令 cache 访问（即轨迹中“I”起始的行）)。 “L”表示数据加载，（L 我们可以简单的认为对某一个地址寄存器的访问）。 “S”表示数据存储，（S 简单的认为对寄存器的访问）。“M”表示数据修改（即数据存储之后的数据加载）。每个“I”前面都没有空格。每个“M”，“L”和“S”之前总是有空格。address 指定一个 32 位的十六进制存储器地址。size 指定操作访问的字节数。

在知晓了这个关键信息后，即可对目标文件中的访存指令进行读取。经过提示，我选择使用 fscanf()函数。

```
    // define parameters
    char operation;
      int addr;
      while (fscanf(file, " %c %x%*c%*d", &operation, &addr) != EOF) {
       //the address field specifies a 64-bit hexadecimal memory access
        if (operation == 'I') {
          continue; //we are only interested in data cache performance
        }
        simulate(addr);
        if ('M' == operation) {
```

```
            simulate(addr);
        }
    }
```

其中的simulate()函数为模拟cache行为的函数，即遵循LRU策略进行替换。

```
void simulate(int addr)
{
    int tag = ((addr >> (set_index_bits + block_bits)) & 0xffffffff);
    int set_index = ((addr>>block_bits)&(0x7fffffff >> (31-set_index_bits)));

    //select set for set[set_index]
    set_t *set = &cache.sets[set_index];

    /*check if cache hit*/
    for (int i = 0; i < cache.line_num; i++) {
        line_t* line = &set->lines[i];

        // Check if the cache line is valid
        if (!(line->valid)) {
            continue;
        }
        // Compare tag bits
        if ((line->tag) != tag) {
            continue;
        }

        /*cache hit*/
        hit_count++;
        update_cache(set, i); // i is the number in lines
        return;
    }

    /*cache miss*/
    miss_count++;

    /*check for cache empty line*/
```

```
    for (int i = 0; i < cache.line_num; i++) {
        line_t* line = &set->lines[i];

        if (line->valid) {
            continue;
        }

        line->valid = true;
        line->tag = tag;
        update_cache(set, i); //i is the number in lines
        return;
    }

    /*neither cache hit or miss, need to evict*/
    eviction_count++;

    /*look for least recently used cache line*/
    for (int i = 0; i < cache.line_num; i++) {
        line_t* line = &set->lines[i];
        if (line->time) {
            continue;
        }
        line->valid = true;
        line->tag = tag;
        update_cache(set, i);
        return;
    }
}
```

由于采用的是 LRU 策略，因此我们需要持续更新每个 cache 行的 time 值，为此需要编写一个 update 函数，更新选取策略采用行号扫描即可。

```
/*update_cache based on line number*/
void update_cache(set_t *set, size_t line_number) {
    line_t *line = &set->lines[line_number];
    for (int i = 0; i < cache.line_num; i++) {
        line_t *temp = &set->lines[i];
```

```
            if (!(temp->valid)) {
                continue;
            }
            if ((temp->time) <= (line->time)) {
                continue;
            }
            --(temp->time); //time: [0, E)
        }
        (line->time) = (cache.line_num - 1); //set time to recently used
    }
```

以上即为 cache 模拟器的基本实现。

1.3.2 矩阵转置优化

分析本题目要求可知，需要创建一个新的矩阵 B[N][M]用以存储转置后的矩阵，并且使函数调用过程中对 cache 的不命中数 miss 尽可能少。

首先根据实验文档可以得到已经给定了 cache 的参数 s=5，b=5，E=1，即 cache 的尺寸为 32sets/32blocks/1B，每组中仅有一行，一行包含 32 块，块大小为 1B，即可存储 8 个整型变量。随后，可以根据这些参数对地址进行划分，低 5 位作为块偏移（block offset），6~10 位作为组索引（set index），行索引则不需要，其余部分均作为 tag 处理.

由于整个 cache 仅能存放 32x8 个整型变量，相较于矩阵的尺寸而言偏小，因此在转置过程中会出现 cache 脱靶现象。

（1）32x32 矩阵

如果使用最朴素的转置函数，即按行遍历 A 矩阵，依次给 B 矩阵对应位置元素赋值的方式，则会因为赋值的即时性导致脱靶。这里地址的格式采用 tag|set index|block index 来表示。由于一个 cache 行仅能读取 8 个整型数字，因此，当需要写入 B[8][0]时，则需要读取 A[0][8]，而第一次进行的一行读取仅包括 A[0][0~7]，此时便出现脱靶现象，需要从内存中载入 A[0][8~15]，以此类推，当 cache 资源消耗完后，转置的进度仍进行的不多，仅载入了部分矩阵，此时便开始频繁的脱靶重载，使 miss 数急剧升高。

因此我们需要关注矩阵转置进行中的局部性原理，以提高函数效率，降低 miss 数。首先可以关注到 cache 的参数为 32sets/32blocks/1B，故一行可以存储 8 个整型变量。已知 cache 的更新策略为整行更新，故可以利用 cache 结构模拟矩阵转置中的对称性。举例而言，每次处理矩阵的 8x8 部分，使一行对应的元素得到充分利用，当处理左上角的 8x8 矩阵时，首先加载 A[0][0~7]，使持续写入到 B[7][0]时不出现驱逐，随后加载 A[1][0~7]，并写入 B[0~7][1]。对于 32x32 矩阵

来说，每 8 行才会出现组索引重复，因此这种分块方式是合理的，即按对角线对称的 8x8 子矩阵。

在对角线上的子矩阵则需要特殊处理，原因如下：在内存中 A[M][N]与 B[M][N]为连续存储，且 32x32 矩阵恰好是 cache 大小的整数倍，因此可以预见到 A 与 B 中下标相同的元素其 set index 与 block offset 均相同，即在 cache 中映射到同一组同一行，此时对对角线上的元素进行原地转置会造成 miss。所以对对角线上的元素需要进行特殊处理。

（2）64X64 矩阵

第二个要求为 64x64 矩阵。此处不能想当然觉得还能使用分块为 8x8 的子矩阵进行转置操作，原因如下：由于整个 cache 的大小为 32x32B，因此在 64x64 矩阵中每隔 4 行便会出现索引重复。此时如果继续使用 8x8 子矩阵，则在子矩阵内部就会出现驱逐情况。例如，写入 B[4][0]时，会加载 A[0][4]，而这两个元素对应的 cache 组号相同，即出现驱逐现象。

解决方案：为了充分利用 cache 空间，仍然可以采用 8x8 的分组方式，不过再对 8x8 矩阵进行处理的时候可以进一步细分为 4x4，以规避组号重复问题。在具体实现过程中根据组号重复规律进行操作。

以左上角两个 8x8 矩阵的转置为例（注意不是最左上角的 8x8，因为对角线与原地转置问题会较为复杂），后续操作类似，在循环中处理，首先创建临时变量用以存储数据，此处存储 A[8][0~7]，分别存入 temp0~temp7 中，可知这 8 个数字填充了对应的一个 cache 行（假设其组号为 0）。

随后将 A[8][4~7]进行转置操作，存入 B[4~7][8]中，可以注意到，A[8~12][0~8]与 B[0~8][8~12]这两块在缓存中的组号不同，因此可以顺利完成这 4 个数的转置。

后续需要进行的一个操作比较重要，即对一部分数据进行暂存。若现在直接将 A[8][0~3]存入 B 中，则需要访问 B[0~3][8]，而这四个数据对应着 4 个 cache 行，set index 分别为 1|9|17|25，这样做会驱逐已经缓存的四行数据 B[4~7][8]，后续依照此原则转置则会出现大量 miss 现象。因此，需要使用 B[4~7][8]中的空闲部分暂存 A[8][0~3]的数据，为了对称，方便循环操作，存入 B[4~7][12]中，如此循环，直到处理完这个 4x4 方格，这时可以看到，B[4~7][12~15]这一 4x4 区域实际存放了需要转置到 B[0~3][8~11]的数据。

随后对 A[12~15][4]进行处理，读入临时变量，此时会将上四行的 cache 缓存驱逐。在此时间点处理这部分的原因是，这一部分正对应着之前暂存的一部分 B[4][12~15]，需要将其对应的块读入并写入该部分，写入前，先将其暂存的值存入临时变量。可以放心读入 A[12~15][4]的原因是，其上 8x4 空间内的数据已得到了充分利用，一部分已经转置完成，另一部分正被暂存。读取完成后，写入 B[4][12~15]。

进行完这步操作后，B[4][12~15]转置完成，其中暂存的变量也已被存至临时变量。这代表着 B[4][8~15]区域，即一个 cache 行代表的区域可以被释放，同时也代表着解放了 B[0][8~15]的使用权，在临时变量数量有限的情况下是难能可贵的，因此马上着手利用这一部分空间。

目前还剩 A[8~15][0~3]部分未转置到 B 上，但是 A[8~11][0]这 4 个值已经暂存在了临时变量中，而 A[12~15][0~3]这一块又在缓存中，且与其目的地不冲突，因此直接进行后续的搬运即可。先将临时变量代表的 A[8~11][0]赋给 B[0][8~11]，再将缓存区中的 A[12~15][0]赋值给 B[0][12~15]，即完成一个 cache 行的工作。如此循环，即可将最后两部分转置完成。

这仅是搬运一个 8x8 方块的工作，为了完成整个矩阵的转置，还需要循环遍历每个 8x8 方块。对于对角线元素的处理本来是想做的，但是发现不处理已经可以达到满分要求，故没有实现。

（3）61x67 矩阵

这是一个不规整的长方形矩阵，可以预见到转置过程中出现的 miss 数也会比较多，因为 cache 行不能恰好完整映射矩阵的行，所以题目的要求也较为宽松。因此，我仅采用了正方形子矩阵分块处理，并测试了子矩阵尺寸对应的 miss 数。

子矩阵尺寸	16x16	17x17	18x18	19x19
Miss 数	1848	1820	1816	1835

可以观察到，在子矩阵尺寸选取 18x18 时，miss 数达到极小值 1816（实际上也是是最小值），因此选取这个结果作为最终答案。边界处理不需要额外的考虑，仅考虑是否越界即可。

1.4. 实验结果和分析

1.4.1 模拟 cache

在本地使用 test-csim 测试如下。

```
chitanda@chitandaeru:~/csapp/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 1.1 模拟 cache 本地测试结果

在 educoder 上也可顺利过关。

1.4.2 矩阵转置优化

分别测试的结果如下。

```
chitanda@chitandaeru:~/csapp/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:1765, misses:288, evictions:256

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:869, misses:1184, evictions:1152

Summary for official submission (func 0): correctness=1 misses=288

TEST_TRANS_RESULTS=1:288
```

图 1.2 32x32 矩阵测试结果

```
chitanda@chitandaeru:~/csapp/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:10097, misses:1172, evictions:1140

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:3473, misses:4724, evictions:4692

Summary for official submission (func 0): correctness=1 misses=1172

TEST_TRANS_RESULTS=1:1172
```

图 1.3 64x64 矩阵测试结果

```
chitanda@chitandaeru:~/csapp/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:6331, misses:1848, evictions:1816

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:3755, misses:4424, evictions:4392

Summary for official submission (func 0): correctness=1 misses=1848

TEST_TRANS_RESULTS=1:1848
```

图 1.4 61x67 矩阵测试结果

```
chitanda@chitandaeru:~/csapp/cachelab-handout$ python2 ./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)       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


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         288
Trans perf 64x64           8.0         8        1172
Trans perf 61x67          10.0        10        1848
          Total points    53.0        53
```

图 1.5 运行测试脚本结果

在 educoder 上亦能通过。

序号	姓名	学号	分班	作业状态	最终成绩	完成时间	实训总耗时	通关情况	评测次数	操作
1	张希哲	U202015311	计科2001-2003...	按时通关	100.0	--	15分钟 19秒	2/2	5	查看

图 1.6 educoder 通过截图

2. 总结和体会

完成 Cachelab 实验后，我对计算机体系结构有了更深入的了解，并且收获了很多经验和体会。在完成实验的过程中，我深刻体会到了计算机体系结构的复杂性和挑战性。尤其是在实现不同的缓存算法时，我需要考虑到不同的指令访存模式、缓存的大小和关联度等因素，这让我深刻认识到了计算机体系结构的复杂性。

总的来说，完成 Cachelab 实验是一项非常有价值的经历。我学会了如何处理计算机系统结构中的复杂问题，同时也提高了自己的编程技能和思考能力。我相信这些经验和技能将对我日后的编程工作有很大的帮助。

3. 对实验课程的建议

实验文档可以细化一点，便于同学们在本地进行调试。例如在命令行中出现的可视化结果开关-v 函数，可以提前编写好给同学们。

同时可以介绍一些内存追踪的工具，以便于更好的理解矩阵转置过程中出现 miss 的原因，以及具体 miss 次数的计算过程。如此可以使编写完转置逻辑后对其运行机理有更清晰的理解。