

目 录

1.	第一部分：Cache 模拟器	3
1.1.	实验目的与要求	3
1.2.	实验环境	3
1.3.	实验思路	3
1.4.	实验结果和分析	8
2.	第二部分：矩阵转置优化	9
2.1.	实验目的与要求	9
2.2.	实验环境	9
2.3.	实验思路	10
2.4.	实验结果和分析	13
3.	总结和体会	14
4.	对实验课程的改进建议	14

1. 第一部分：Cache 模拟器

1.1. 实验目的与要求

- 本次 cache 实验目的在于加深 Cache 缓存组成结构对 C 程序性能的影响的理解。
- 任务：在 csim.c 提供的程序框架中，编写实现一个 Cache 模拟器：
 - 输入：内存访问轨迹
 - 操作：模拟缓存相对内存访问轨迹的命中/缺失行为
 - 输出：命中、缺失和（缓存行）淘汰/驱逐的总数
 - 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（而不是代码中固定大小的值）来为模拟器中数据结构分配存储空间。
 - 由于实验仅关心数据 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）
 - 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——因此可忽略访问轨迹中给出的访问请求大小
 - main 函数最后必须调用 printSummary 函数输出结果，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

- 操作系统：Ubuntu 16.04
- 编译器：gcc
- 内存调试：valgrind

1.3. 实验思路

仔细观察 csim.c 文件，发现大体框架已经给出，需要实验的就是以下函数：

```
1. freeCache()  
2. accessData()  
3. replayTrace()
```

其余的函数如 initCache 等都已经给出实现。要实现以上三个函数，我们必须清楚 cache 的结构是怎么定义的，这也在文件中给出：

```

1. typedef struct cache_line {
2.     char valid;
3.     mem_addr_t tag;
4.     unsigned long long int lru;
5. } cache_line_t;
6.
7. typedef cache_line_t* cache_set_t;
8. typedef cache_set_t* cache_t;

```

这个 cache 的结构也可以由 initialCache 函数推出。如图 1.1 cache 结构（cmu 课件）。

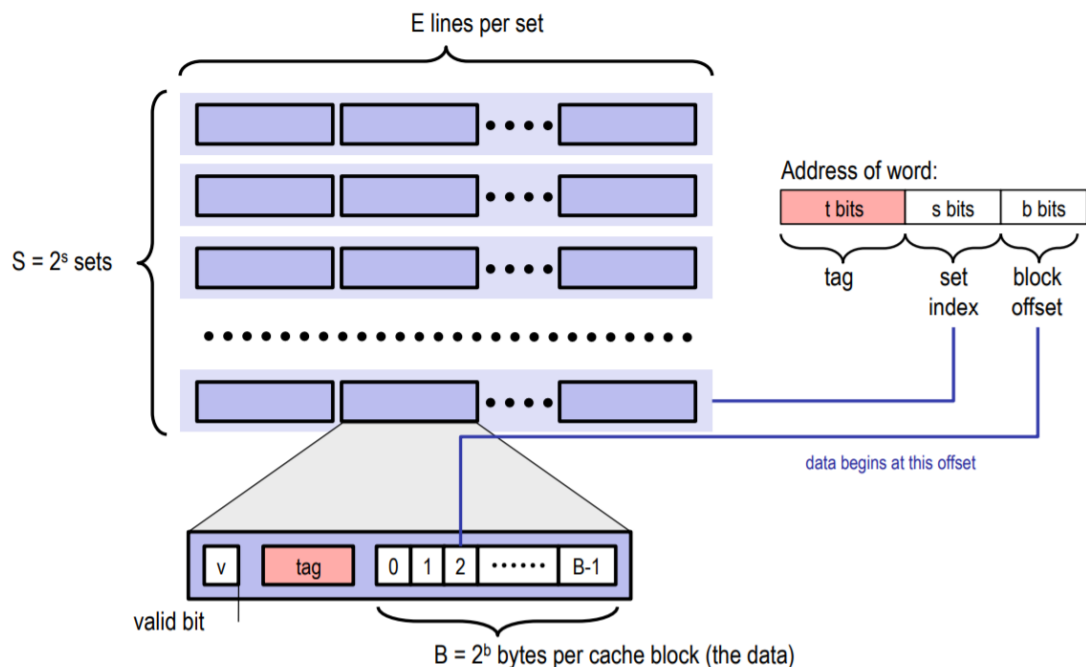


图 1.1 cache 结构

下面就给出三个函数的详细设计：

1. freeCache

freeCache 这个函数十分好些，由 initialCache 可知 cache 其实就是一个二维数组。所以我们只需先释放 cache[i] 的内存，之后释放掉 cache 的内存即可完成二维数组的内存释放。在这里就不给出流程图而是直接贴出代码：

```

1. int i;
2. for(i = 0; i < S; i++){
3.     free(cache[i]);
4. }

```

```
5. free(cache);
```

2. accessData

`accessData` 模拟的就是读写 `cache` 的功能。首先它需要一个参数传递地址信息。根据参数可以确定 `tag`、`set index`，根据 `set index` 找到 `cache` 组，然后遍历组内每个 `cacheline`，看是否能找到 `tag` 相匹配并且 `valid` 位有效。如果找到，则 `hit` 次数加一，更新该 `cacheline` 的 `lru` (`lrucounter` 加 1)。否则 `miss` 次数加一，然后遍历找到该 `cache` 组内 `lru` 最小的哪个 `cacheline`，如果它的 `valid` 位有效，这说明发生了替换，此时 `eviction` 计数加一。然后使此 `cacheline` 的 `valid` 位置 1，`tag` 位为参数的 `tag` 位，`lru` 在 `lrucounter` 的基础上加一。在这里的 LRU 替换算法有点不同，它的全局变量 `lrucounter` 相当于一个标记，一个指针。总是指向最近使用的 `cacheline`。而其他的 `cacheline` 的 `lru` 成员的值也可以按照大小顺序排成一列，相当于一个队列，这样也可以实现 `lru` 的功能。如图 1.2 `lru` 算法，图 1.3 `accessData` 流程图。

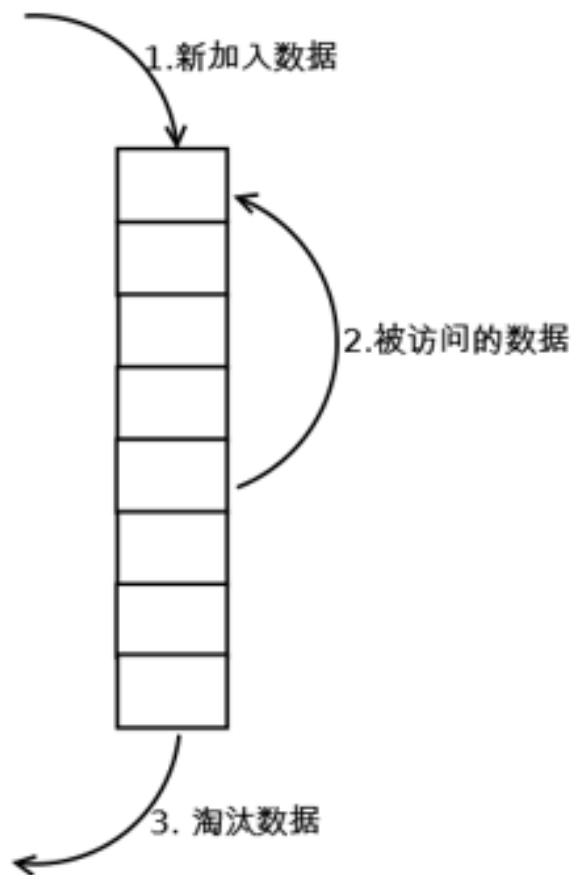


图 1.2 `lru` 算法

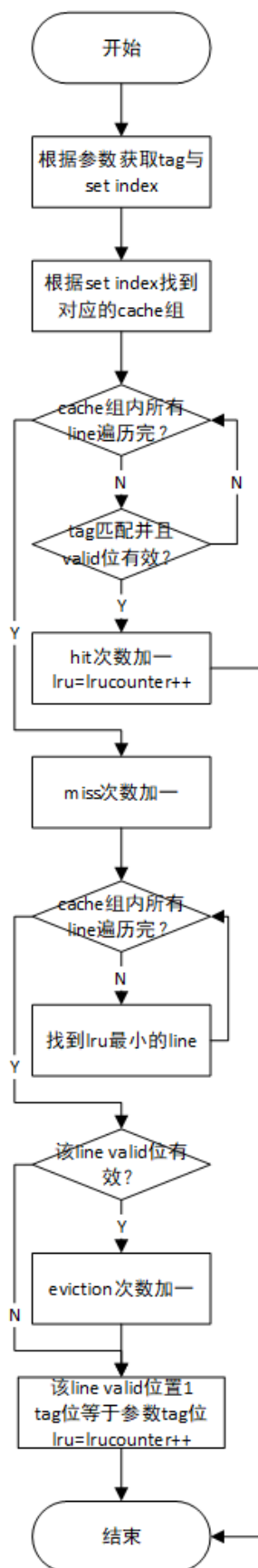


图 1.3 accessData 流程图


```
32.         }
33.         break;
34.     default:
35.         break;
36.     }
37. }
38.
39.     fclose(trace_fp);
40. }
```

1.4. 实验结果和分析

只需要执行 `make` 命令，以及 `./test-csim` 命令即可完成对 `csim` 的测试，如图 1.4 `csim` 测试结果。

```
may@may-VirtualBox:~/Share/cacheLab$ ./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
may@may-VirtualBox:~/Share/cacheLab$
```

图 1.4 `csim` 测试结果

由上述实验结果可知，本次 `csim` 实现的还是很成功的，对于它给出的测试用例都得到了理想的输出。

2. 第二部分：矩阵转置优化

2.1. 实验目的与要求

- 本次 cache 实验目的在于加深 Cache 缓存组成结构对 C 程序性能的影响的理解。
- 任务：在 trans.c 中编写实现一个矩阵转置函数 transpose_submit，要求其在参考 Cache 模拟器 csim-ref 上运行时对不同大小的矩阵均能最小化缓存缺失的数量
- 实现要求：
 - 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。
 - 原因：实验测试代码不能/不应计数栈的引用访问，而应将注意力集中在对源和目的矩阵的访问模式上
 - 不允许使用递归。如果定义和调用辅助函数，在任意时刻，从转置函数的栈帧到辅助函数的栈帧之间最多可以同时存在 12 个局部变量。
 - 例如，如果转置函数定义了 8 个局部变量，其中调用了一个使用 4 个局部变量的函数，而其进一步调用了一个使用 2 个局部变量的函数，则栈上总共将有 14 个变量，则违反了本规则。
 - 转置函数不允许改变矩阵 A，但可以任意操作矩阵 B。
 - 不允许在代码中定义任何矩阵或使用 malloc 及其变种。
- 评分：
 - 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 8 分，如果 m>600 得 0 分，对其他 m 得(600-m)*8/300 分。
 - 64×64：如果 m<1300 得 8 分，如果 m>2000 得 0 分，对其他 m 得(2000-m)*8/700 分。
 - 61×67：如果 m<2000 得 10 分，如果 m>3000 得 0 分，对其他 m 得(3000-m)*10/1000 分。

2.2. 实验环境

- 操作系统： Ubuntu 16.04
- 编译器： gcc

➤ 内存调试: valgrind

2.3. 实验思路

首先要了解到要充分利用 cache 的性能, 需要将矩阵存入 cache, 但是 cache 显然不能一次性存入矩阵的全部数据, 因此将矩阵分块, 使得 cache 的 miss 次数尽量减少。本次实验 cache 的大小为 32 行, 每行 32 字节。并且只能使用 12 个 int 变量。

1. 32×32

cache 一行 32 字节, 可以容纳下 8 个 int。而且这个 cache 可以容纳下矩阵的前 8 行, 所以分块的话采取 8×8 是比较合适的。对于非对角线上的块来说, 由于写的时候会把之前读的块给覆盖掉, 所以此时读取次数会翻倍。这样来说 miss 次数理论上大概为: $(8+8) \times 12 + (8+16) \times 4 = 288$, 符合题目的要求。代码如下:

```
1. for(i=0;i<N;i+=8){
2.     for(j=0;j<M;j+=8){
3.         for(k=i;k<i+8&& k<N;k++){
4.             temp1=A[k][j];
5.             temp2=A[k][j+1];
6.             temp3=A[k][j+2];
7.             temp4=A[k][j+3];
8.             temp5=A[k][j+4];
9.             temp6=A[k][j+5];
10.            temp7=A[k][j+6];
11.            temp8=A[k][j+7];
12.
13.            B[j][k]=temp1;
14.            B[j+1][k]=temp2;
15.            B[j+2][k]=temp3;
16.            B[j+3][k]=temp4;
17.            B[j+4][k]=temp5;
18.            B[j+5][k]=temp6;
19.            B[j+6][k]=temp7;
20.            B[j+7][k]=temp8;
21.        }
22.    }
23. }
```

2. 64×64

此题的难度比较大,经过测试,简单的分块已经不能满足题目要求的 1300 次了。如果继续采用 8×8 分块,由于 cache 只能容纳矩阵的前四行,所以 miss 次数会增加很多。

如果简单地只用 4×4 分组呢,那么 cache 的空间就会被浪费,最后测试出来 miss 次数也是远远达不到题目所要求的 1300 次。

为了解决这个问题,我在网络上搜集了大量的资料,最后参考了网上的一种解题思路。如图 2.1 64×64 解题思路(来源网络)。

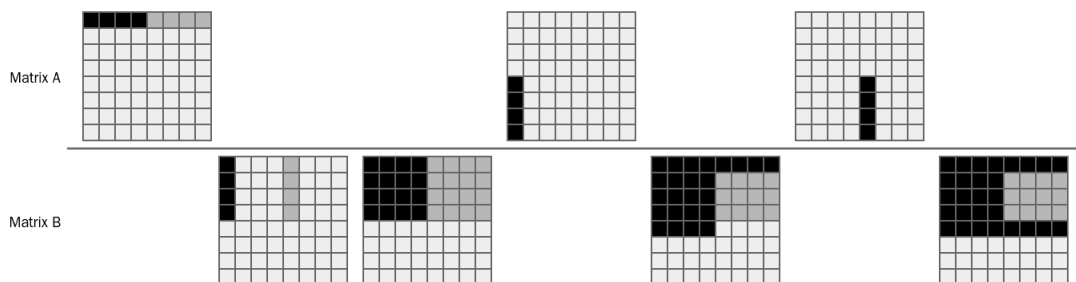


图 2.1 64×64 解题思路

下面给出了上图的解释:

1.将 A 矩阵的上半部分分成两部分独立的矩阵来看待,进行转置。进行这一操作之后, B 矩阵左上角黑色部分的 16 个块的位置是正确的,而右上角灰色部分的 16 个块,只需要将它们整体移到 B 矩阵的左下角,它们的位置就正确了。

2.将 A 矩阵的左下角一列的 4 个块进行转置,同时把原来灰色部分上面的一行 4 个块移到 B 矩阵左下角的第一行前四个块。

3.将 A 矩阵的右下角一列的 4 个块进行转置。

4.重复 2,3 步骤直到矩阵完成转置。

这个解题思路我后来考虑了很久,它的优点是充分利用了 cache 的空间(保持了 8×8 的分块结构)。同时并不要求一次就把 A 矩阵中的块全部转置到 B 矩阵中的正确位置,之后可以对 B 矩阵进行单独的操作,使其位置正确。

这个解题思路减少 miss 次数的关键步骤是第 2 步,这步用了 int 型的中间变量,将 A 矩阵的左下角一列的 4 个块, B 矩阵右上角一行的 4 个块全部用中间变量保存,然后进行变换。B 右上角的块读取时已经放入了 Cache,所以 A 矩阵的块写入时不会产生 miss 操作。

这种方案的 miss 次数比较难算,第一步中对于非对角块, miss 次数为 $(4+4)$,对于对角块, miss 次数为 $(4+7)$ 。第二步中,对于非对角块, miss 次数为 $(4+4)$,对于对角块, miss 次数为 $(4+3+3+4+4)$ 。第三步中,对于非对角块, miss 次数为 0,对于对角块 miss 次数为 $(4+7)$ 。这样总的 miss 次数应该是 $8 \times (11+18+11) + 56 \times (8+8) = 1216$ 。可以满足题目要求。代码如下:

```
1. for(i=0;i<N;i+=8){
```

```
2.     for(j=0;j<M;j+=8){
3.         for(k=j;k<j+4;k++){
4.             temp1=A[k][i];
5.             temp2=A[k][i+1];
6.             temp3=A[k][i+2];
7.             temp4=A[k][i+3];
8.             temp5=A[k][i+4];
9.             temp6=A[k][i+5];
10.            temp7=A[k][i+6];
11.            temp8=A[k][i+7];
12.
13.            B[i][k]=temp1;
14.            B[i][k+4]=temp5;
15.            B[i+1][k]=temp2;
16.            B[i+1][k+4]=temp6;
17.            B[i+2][k]=temp3;
18.            B[i+2][k+4]=temp7;
19.            B[i+3][k]=temp4;
20.            B[i+3][k+4]=temp8;
21.        }
22.        for(k=i;k<i+4;k++){
23.            temp1=B[k][j+4];
24.            temp2=B[k][j+5];
25.            temp3=B[k][j+6];
26.            temp4=B[k][j+7];
27.            temp5=A[j+4][k];
28.            temp6=A[j+5][k];
29.            temp7=A[j+6][k];
30.            temp8=A[j+7][k];
31.
32.            B[k][j+4]=temp5;
33.            B[k][j+5]=temp6;
34.            B[k][j+6]=temp7;
35.            B[k][j+7]=temp8;
36.            B[k+4][j]=temp1;
37.            B[k+4][j+1]=temp2;
38.            B[k+4][j+2]=temp3;
39.            B[k+4][j+3]=temp4;
40.        }
41.        for(k=i+4;k<i+8;k++){
42.            temp1=A[j+4][k];
43.            temp2=A[j+5][k];
44.            temp3=A[j+6][k];
45.            temp4=A[j+7][k];
```

```

46.
47.         B[k][j+4]=temp1;
48.         B[k][j+5]=temp2;
49.         B[k][j+6]=temp3;
50.         B[k][j+7]=temp4;
51.     }
52. }
53. }

```

3. 61×67

此题的要求比较宽松，miss 次数小于 2000 即可完成题目要求。因此在这里尝试了不同的分块方案，最后也是采用 16×16 的分块可以满足要求。代码如下：

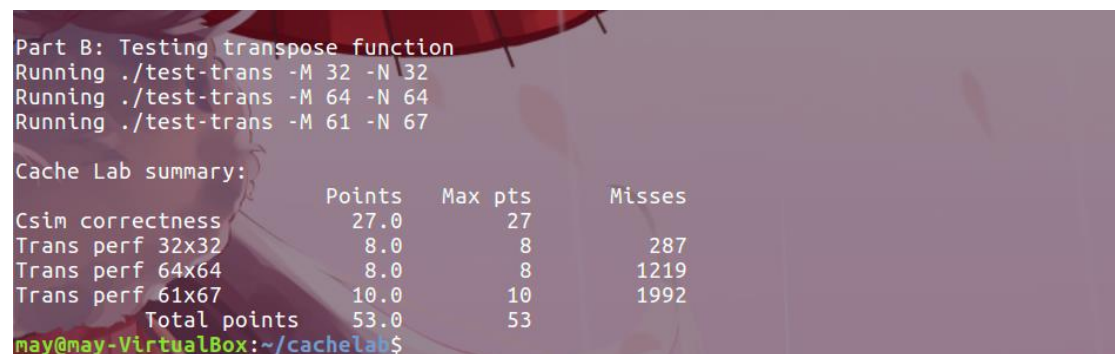
```

1. for(i=0;i<N;i+=16){
2.     for(j=0;j<M;j+=16){
3.         for(k=i;k<i+16&& k<N;k++){
4.             for(h=j;h<j+16&& h<M;h++){
5.                 B[h][k]=A[k][h];
6.             }
7.         }
8.     }
9. }

```

2.4. 实验结果和分析

只需要执行 make 命令，以及 ./driver.py 命令即可完成所有的测试，如图 2.2 trans 测试结果。



```

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	287
Trans perf 64x64	8.0	8	1219
Trans perf 61x67	10.0	10	1992
Total points	53.0	53	

```

may@may-VirtualBox:~/cacheLab$

```

图 2.2 trans 测试结果

由上述实验结果可知，本次矩阵转置实现的还是很成功的，对于它给出的测试用例都得到了理想的输出，都小于它的要求 miss 次数。

3. 总结和体会

第一个实验还是比较简单的，只要清楚 cache 的结构是什么样的就可以写出来。同时自己模拟的更新 cache 的功能是基于 lru 算法上实现的。通过自己编写程序模拟 cache，更好地了解了 cache 地址映射的关系，以及 cache 的工作方式。了解这些更有利于编写出对 cache 友好的代码。

第二个实验的难度比较大，平时我们编写程序很少考虑到 cache，而这个实验通过一个矩阵转换的例子，将代码的性能通过 cache 的 miss 次数量化了。这个实验的三个测试中第二个测试的难度是最大的，其余的测试只要尝试不同的分块即可完成，同时考虑到 cache 的容量以及矩阵的容量，初始分块也就能够比较容易地确定了。本次实验完成后，能够更深入地理解 cache，并且也知道了编写 cache 友好的代码是很重要的，而这条路还有很长。

4. 对实验课程的改进建议

建议本门课程的第二个实验在 ppt 上给出更详细的说明，也可以提供一些思路。第二个实验的难度较大，如果没有参考网上资料的话，是很难自己思考出有效地解题方法的。