

CacheLab Report By 2019201408陈志朋

Part - A

1) 对指令的处理

根据文档提示，*I*指令可以无需进行处理；*S*和*L*的处理类似，会对cache进行一次查询，如果没有命中的话会将所需的内容加载到cache中；*M*指令可以分解为*S*指令加上*L*指令。同时，文档中限定了每次读取了字节不会分布在两个块中，所以读取的字节长度也不需要进行处理。

代码如下：

```
1 void solve() {
2     FILE *F = freopen (file, "r", stdin); /* 打开所需文件 */
3
4     init_cache();
5     char inst[5], addr[100];
6     long long tag;
7     int l, r, sid;
8     while (scanf ("%s", inst) != EOF) { /* 读入指令 */
9         scanf ("%s", addr); /* 读入地址 */
10        step += 1; /* 更新时间戳 */
11        for (l = 0, r = 0; addr[r] != ','; ++r);
12        /* 得到有效的地址在字符串的起始位置和终止位置 */
13        htoi(addr, l, r, &tag, &sid); /* 将16进制的地址转化为二进制
14        */
15        if (inst[0] == 'I') continue; /* 不对I指令进行处理 */
16        else if (inst[0] == 'L') { /* 处理L指令 */
17            if (flag) printf ("%s %s", inst, addr); /* 处理-v参
18            数 */
19            load_memory(tag, sid); /* 进行一次内存加载 */
20            if (flag) puts(""); /* 处理-v参数 */
21        }
22        else if (inst[0] == 'S') { /* 处理S指令 */
23            if (flag) printf ("%s %s", inst, addr); /* 处理-v参
24            数 */
25            store_memory(tag, sid); /* 进行一次内存存储 */
26            if (flag) puts(""); /* 处理-v参数 */
27        }
28        else if (inst[0] == 'M') { /* 处理M指令 */
29            if (flag) printf ("%s %s", inst, addr); /* 处理-v参
30            数 */
31            load_memory(tag, sid); /* 进行一次内存加载 */
32            store_memory(tag, sid); /* 进行一次内存存储 */
33            if (flag) puts(""); /* 处理-v参数 */
34        }
35    }
```

```

31     }
32 }
33
34 fclose (F); /* 关闭文件 */
35 }

```

2) 对cache的模拟

可以使用一个 $2^s \times E$ 的二维数组实现。由于 s 和 E 是变量，所以不能使用二维数组，会缺乏可拓展性。这里使用指针的指针来代替二维数组，实现了一个大小可变的二维数组，只需每次使用malloc函数申请内存即可。在模拟cache的同时，需要开一个相同大小的数组，记录cache中每个位置的时间戳。

cache初始化的代码如下：

```

1  long long **cache; /* 储存每个位置的tag */
2  long long **tim; /* 储存对应位置的时间戳 */
3
4  void init_cache() {
5      cache = malloc((1 << s) * sizeof(long long *)); /* 先开一个
   行数组 */
6      for (int i = 0; i < (1 << s); ++i) {
7          cache[i] = malloc(E * sizeof(long long));
8      } /* 对每一行开相应的列 */
9      for (int i = 0; i < (1 << s); ++i) {
10         for (int j = 0; j < E; ++j)
11             cache[i][j] = 0;
12     } /* 将整个cache数组清零 */
13
14     tim = malloc((1 << s) * sizeof(long long *)); /* 先开一个行数组 */
15     for (int i = 0; i < (1 << s); ++i) {
16         tim[i] = malloc(E * sizeof(long long));
17     } /* 对每一行开相应的列 */
18     for (int i = 0; i < (1 << s); ++i) {
19         for (int j = 0; j < E; ++j)
20             tim[i][j] = 0;
21     } /* 将整个cache数组清零 */
22 }

```

3) 处理地址

对于一个地址，由于每个十六进制位对应了4个二进制位，所以从最低位开始，用if-else语句实现十六进制到二进制的转化。转化为二进制后，第 b 位至第 $(b + s - 1)$ 位对应的是行号，剩下的是tag，这里直接将tag储存为十进制进行储存。

地址转化的代码如下：

```

1  void htob(char *addr, int l, int r, long long *tag, int *sid) {
2      (*tag) = 0; (*sid) = 0;
3      for (int i = 0; i < 64; ++i) bin[i] = 0;

```

```

4     for (int i = r - 1, p = 0; i >= 1; --i) {
5         if (addr[i] == '1' || addr[i] == '3' || addr[i] == '5'
6         || addr[i] == '7' ||
7             addr[i] == '9' || addr[i] == 'b' || addr[i] == 'd'
8         || addr[i] == 'f')
9             bin[p] = 1; /* 处理第1位 */
10            p += 1;
11
12            if (addr[i] == '2' || addr[i] == '3' || addr[i] == '6'
13            || addr[i] == '7' ||
14                addr[i] == 'a' || addr[i] == 'b' || addr[i] == 'e'
15            || addr[i] == 'f')
16                bin[p] = 1; /* 处理第2位 */
17                p += 1;
18
19                if (addr[i] == '4' || addr[i] == '5' || addr[i] == '6'
20                || addr[i] == '7' ||
21                    addr[i] == 'c' || addr[i] == 'd' || addr[i] == 'e'
22                || addr[i] == 'f')
23                    bin[p] = 1; /* 处理第3位 */
24                    p += 1;
25
26                    if (addr[i] == '8' || addr[i] == '9' || addr[i] == 'a'
27                    || addr[i] == 'b' ||
28                        addr[i] == 'c' || addr[i] == 'd' || addr[i] == 'e'
29                    || addr[i] == 'f')
30                        bin[p] = 1; /* 处理第4位 */
31                        p += 1;
32            }
33
34            for (int i = b + s - 1; i >= b; --i)
35                (*sid) = (*sid) * 2 + bin[i]; /* 计算该地址对应的行号 */
36
37            for (int i = 63; i >= b + s; --i)
38                (*tag) = (*tag) * 2 + bin[i]; /* 计算该地址对应的tag */
39        }

```

4) 加载内存

对cache查询时，先找到地址对应的行，然后枚举这行中的每一个格子。如果是空格，就进行标记；如果是当前地址的tag，那么该地址已经储存在cache中，退出查找。否则，记录一个时间戳最小的格子。当所有格子都枚举过之后，如果有找到，则为hit。否则为miss，若没有空格子，需要在加上eviction。最后更新格子的tag值和时间戳。

加载内存的代码如下：

```

1 void load_memory(long long tag, int sid) {
2     int pos = 0;
3     bool isfree = false, isfind = false;
4     for (int i = 0; i < E; ++i) {
5         if (cache[sid][i] == tag && tim[sid][i] != 0) {

```

```

6         isfind = true;
7         tim[sid][i] = step;
8         break;
9     } /* 当前地址可以在cache中找到 */
10    if (tim[sid][i] == 0) isfree = true; /* 当前格是空格 */
11    if (tim[sid][i] < tim[sid][pos]) pos = i; /* 找一个时间戳
最小的，空格时间戳为0 */
12    }
13    if (isfind == true) {
14        hit += 1;
15        if (flag) printf (" hit");
16    } /* cache命中 */
17    else {
18        miss += 1;
19        if (flag) printf (" miss"); /* 不命中 */
20        if (!isfree) {
21            evic += 1;
22            printf (" eviction");
23        } /* 需要更新cache */
24        cache[sid][pos] = tag;
25        tim[sid][pos] = step; /* 对cache进行更新 */
26    }
27 }

```

5) 内存储存

直接调用加载内存的函数即可。

6) 处理命令行参数

使用argc和argv将命令行参数传入到程序中，枚举每一个参数进行处理。

代码如下：

```

1  int main(int argc, char *argv[]) {
2      for (int i = 1; i < argc; ++i) { /* 枚举每一个参数 */
3          if (argv[i][1] == 'v') { /* 处理-v */
4              flag = true;
5          }
6          else if (argv[i][1] == 's') { /* 读取s的大小 */
7              s = atoi(argv[i + 1]);
8              i += 1;
9          }
10         else if (argv[i][1] == 'E') { /* 读取E的大小 */
11             E = atoi(argv[i + 1]);
12             i += 1;
13         }
14         else if (argv[i][1] == 'b') { /* 读取b的大小 */
15             b = atoi(argv[i + 1]);
16             i += 1;
17         }

```

```

18         else if (argv[i][1] == 't') { /* 读取输入文件的位置 */
19             file = argv[i + 1];
20             i += 1;
21         }
22     }
23     solve(); /* 调用处理函数 */
24     printSummary(hit, miss, evic); /* 输出结果 */
25     return 0;
26 }

```

Part - B

1) 32×32 的矩阵

由于cache的大小是 $s = 5$, $E = 1$, $b = 5$, 即一共有 $2^5 = 32$ 行, 每行可以储存一个tag, 可以储存32个字节的内容。因此, 矩阵每8行在cache中对应的行数是一样的(第 i 行和第 $i + 8$ 行储存在cache中对应的行数)。所以, 可以对每 8×8 的子矩阵进行分块以最大利用cache。

代码如下:

```

1 void transpose1(int M, int N, int A[N][M], int B[M][N]) {
2     int i, j, p, q;
3     for (i = 0; i < 32; i += 8) {
4         for (j = 0; j < 32; j += 8) {
5             for (p = i; p < i + 8; ++p)
6                 for (q = j; q < j + 8; ++q)
7                     B[q][p] = A[p][q];
8         }
9     }
10 }

```

测试结果如下图所示:

```

2019201408@ics-student:~/cachelab/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

TEST TRANS RESULTS=1:343

```

但是miss的次数为343次, 比最优解还多一点。所以考虑在哪些地方可以降低miss次数。根据cache的大小可以发现, A 数组对应行在cache中的位置和 B 数组相同行的位置一样。对于处理下图标黄块时, 会发生多次的miss。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	Yellow								Blue								White								White							
2																																
3																																
4																																
5																																
6																																
7																																
8																																
9	White								Yellow								Blue								White							
10																																
11																																
12																																
13																																
14																																
15																																
16																																
17	White								White								Yellow								Blue							
18																																
19																																
20																																
21																																
22																																
23																																
24																																
25	Blue								White								White								Yellow							
26																																
27																																
28																																
29																																
30																																
31																																
32																																

可以考虑先处理对角线的块，将上图A数组中每行的黄块先移动到B数组同行的蓝块上。在将B数组的蓝块移动到B数组每行对应的黄块上，这样减少了大量的miss。

代码如下：

```

1 void transpose1(int M, int N, int A[N][M], int B[M][N]) {
2     int i, j, p, q;
3     for (i = 0; i < 32; i += 8) {
4
5         for (p = i; p < i + 8; ++p)
6             for (q = i; q < i + 8; ++q)
7                 B[q][(p + 8) % N] = A[p][q]; /* 16次miss */
8
9         for (p = i; p < i + 8; ++p)
10            for (q = i; q < i + 8; ++q)
11                B[p][q] = B[p][(q + 8) % N]; /* 8次miss */
12    }
13
14    for (i = 0; i < 32; i += 8) {
15        for (j = 0; j < 32; j += 8) {
16            if (i != j) {
17                for (p = i; p < i + 8; ++p)
18                    for (q = j; q < j + 8; ++q)
19                        B[q][p] = A[p][q]; /* 16次miss */
20            }
21            else continue;
22        }
23    }
24 }

```

测试结果如下图所示：

```
2019201408@ics-student:~/cachelab/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:2274, misses:291, evictions:259

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=291

TEST_TRANS_RESULTS=1:291
```

2) 64×64 的矩阵

先尝试 8×8 分块的程序，与上一问的程序类似，测试结果如下图所示，

```
2019201408@ics-student:~/cachelab/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:4482, misses:4739, evictions:4707

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=4739

TEST_TRANS_RESULTS=1:4739
```

发现该算法的测试结果极差。考虑到矩阵大小变大的，所以每4行后在cache中对应的行会重复出现一次。所以每个 8×8 分块中的前4行和后4行会产生冲突，导致了miss次数变多。因此，尝试修改乘 4×4 分块。测试结果如下图所示，

```
2019201408@ics-student:~/cachelab/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:6858, misses:1851, evictions:1819

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=1851

TEST_TRANS_RESULTS=1:1851
```

可以发现， 4×4 分块已经极大的减少了cache冲突的问题，但是由于每块的大小太小，没有充分利用cache的空间，而且块数过多，导致miss的次数仍然很多。

考虑先将整个矩阵按 8×8 分块，每个分块中将前4行和后4行分开处理。

第一步把A数组的前4行的前4个元素和后4个元素分别转置，此处会产生8次miss，如下图所示（上面的为A，下面的为B）：

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64
1	9	17	25	5	13	21	29
2	10	18	26	6	14	22	30
3	11	19	27	7	15	23	31
4	12	20	28	8	16	24	32

注意处理过程中，为了避免cache冲突，可以开8个int型的临时变量来储存A的某一行，可以避免A和B两个数组之间的冲突。

第二步把B的右上角的数拷贝到左下角，然后再将A的左下角拷贝到B的右上角。在这个过程中，可以通过调整赋值顺序利用已经储存到cache中的数据进行操作：先拷贝B数组的右上角到临时变量（4次miss），然后拷贝A的左下角到临时变量（4次miss），接着赋值B的右上角（已经加载在cache中），最后赋值B的右下角（4次miss）。

第三步处理B的右下角。由于第二步中已经加载了A的后四行和B的后四行，所以此处不会产生miss，直接处理即可（使用临时变量防止A、B之间的冲突）。

代码如下：

```
1 void transpose2(int M, int N, int A[N][M], int B[M][N]) {
2     int i, j, p;
3     int a1, a2, a3, a4, a5, a6, a7, a0;
4
5     for (i = 0; i < 64; i += 8) {
6         for (j = 0; j < 64; j += 8) {
7             for (p = 0; p < 4; ++p) {
8                 a0 = A[i + p][j + 0];
9                 a1 = A[i + p][j + 1];
10                a2 = A[i + p][j + 2];
11                a3 = A[i + p][j + 3];
12                a4 = A[i + p][j + 4];
13                a5 = A[i + p][j + 5];
14                a6 = A[i + p][j + 6];
15                a7 = A[i + p][j + 7];
16
17                B[j + 0][i + p] = a0;
18                B[j + 1][i + p] = a1;
19                B[j + 2][i + p] = a2;
20                B[j + 3][i + p] = a3;
21                B[j + 0][i + p + 4] = a4;
22                B[j + 1][i + p + 4] = a5;
```



```

23         B[j + 2][i + p + 4] = a6;
24         B[j + 3][i + p + 4] = a7;
25     }
26
27     for (p = 0; p < 4; ++p) {
28         a0 = B[j + p][i + 4];
29         a1 = B[j + p][i + 5];
30         a2 = B[j + p][i + 6];
31         a3 = B[j + p][i + 7];
32
33         a4 = A[i + 4][j + p];
34         a5 = A[i + 5][j + p];
35         a6 = A[i + 6][j + p];
36         a7 = A[i + 7][j + p];
37
38
39         B[j + p][i + 4] = a4;
40         B[j + p][i + 5] = a5;
41         B[j + p][i + 6] = a6;
42         B[j + p][i + 7] = a7;
43
44         B[j + p + 4][i + 0] = a0;
45         B[j + p + 4][i + 1] = a1;
46         B[j + p + 4][i + 2] = a2;
47         B[j + p + 4][i + 3] = a3;
48     }
49
50     for (p = 4; p < 8; ++p) {
51         a0 = A[i + p][j + 4];
52         a1 = A[i + p][j + 5];
53         a2 = A[i + p][j + 6];
54         a3 = A[i + p][j + 7];
55
56         B[j + 4][i + p] = a0;
57         B[j + 5][i + p] = a1;
58         B[j + 6][i + p] = a2;
59         B[j + 7][i + p] = a3;
60     }
61 }
62 }
63
64 }

```

测试结果如下图所示，

```

2019201408@ics-student:~/cachelab/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:9018, misses:1227, evictions:1195

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=1227

TEST_TRANS_RESULTS=1:1227

```

3) 67×61 的矩阵

由于行和列都不是32的倍数，所以没有太优秀的分块做法。任何分块做法都会余数的地方造成大量的冲突，所以先尝试普通的 8×8 分块做法。代码和 32×32 的第一种相同。

测试结果如下图所示，

```

2019201408@ics-student:~/cachelab/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:6061, misses:2118, evictions:2086

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=2118

TEST_TRANS_RESULTS=1:2118

```

可以发现结果和要求差一点，所以可以尝试修改分块的大小。当分块的大小变大时，每一次可能会多一些冲突，但是总的运行次数变少了。经过测试，当分块大小为 18×18 时，此时测试结果能得到满分。

代码如下，

```

1  const int maxn = 18;
2  void transpose3(int M, int N, int A[N][M], int B[M][N]) {
3      int i, j, p, q;
4
5      for (i = 0; i < 67; i += maxn) {
6          for (j = 0; j < 61; j += maxn) {
7              for (p = i; p < 67 && p < i + maxn; ++p)
8                  for (q = j; q < 61 && q < j + maxn; ++q) {
9                      B[q][p] = A[p][q];
10                 }
11             }
12         }
13     }

```

测试结果如下图所示，

```

2019201408@ics-student:~/cachelab/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:6218, misses:1961, evictions:1929

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=1961

TEST_TRANS_RESULTS=1:1961

```

总结

所有部分均通过测试：

```

2019201408@ics-student:~/cachelab/cachelab-handout$ python2 ./driver.py
Part A: Testing cache simulator
Running ./test-csim

```

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)	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	291
Trans perf 64x64	8.0	8	1227
Trans perf 61x67	10.0	10	1961
Total points	53.0	53	