

# 华中科技大学

## 计算机体系结构

学 生 姓 名：	胡思勛
学 号：	U201514898
专 业：	计算机科学与技术
班 级：	计卓 1501
指 导 教 师：	吴菲

# 目录

<b>第 1 章 Building a Cache Simulator</b>	<b>1</b>
1.1 实验要求 . . . . .	1
1.2 实验设计 . . . . .	2
1.2.1 总体设计 . . . . .	2
1.2.2 详细设计 . . . . .	3
1.3 实验过程 . . . . .	5
1.3.1 实验环境 . . . . .	5
1.3.2 详细步骤 . . . . .	5
1.3.3 测试与分析 . . . . .	7
<b>第 2 章 Optimizing Matrix Transpose</b>	<b>9</b>
2.1 实验要求 . . . . .	9
2.2 实验设计 . . . . .	9
2.2.1 总体设计 . . . . .	9
2.2.2 详细设计 . . . . .	10
2.3 实验过程 . . . . .	13
2.3.1 实验环境 . . . . .	13
2.3.2 详细步骤 . . . . .	13
2.3.3 测试与分析 . . . . .	16
<b>心得体会</b>	<b>17</b>
<b>源代码</b>	<b>17</b>

# 第 1 章 Building a Cache Simulator

整个实验分为两个部分，在第一部分中，需要实现一个缓存模拟器。通过 valgrind 中的 lackey 工具<sup>1</sup>可以得到一个程序几乎所有的内存访问情况，使用如下命令即可获得这些信息：

```
linux> valgrind --log-fd=1 --tool=lackey --trace-mem=yes <program>
```

一个样例输出如下：

```
I 0400d7d4,8  
M 0421c7f0,4  
L 04f6b868,8  
S 7ff0005c8,8
```

输出的格式为“操作 地址, 大小”，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，而数据修改应该被当做一次数据加载加上一次数据存储。内存地址以十六进制的形式给出。

## 1.1 实验要求

此实验要求编写一个 Cache 模拟器，其输入为 Valgrind 输出的内存访问轨迹，输出为与 csim-ref 相同的统计数据。

实验要求：

- 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：

在编写完成后，使用 test-csim 程序进行测试以及评分。Cache 模拟器使用的策略应为 LRU 替换策略。

---

<sup>1</sup><http://valgrind.org/docs/manual/lk-manual.html>

## 1.2 实验设计

### 1.2.1 总体设计

需要修改的文件为 csim.c。由于已经给出了框架，首先观察框架中的代码，其中包含以下函数：

```
1  accessData(mem_addr_t addr)
2  freeCache()
3  initCache()
4  main(int argc, char * argv[])
5  printUsage(char * argv[])
6  replayTrace(char * trace_fn)
```

根据函数名和注释可以得知，initCache 和 freeCache 用于初始化和释放 cache，printUsage 用于打印帮助信息，而 replayTrace 则直接被主函数调用并调用 accessData 来模拟 Cache 的替换过程。而需要修改的函数就是 accessData 函数。

接下来观察 Cache 是如何在 C 语言中被组织并模拟的。首先，文件中定义了如下的结构，来描述 cache line，并定义其指针以及指针的指针分别为 cache set 和 cache。

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

而通过 initCache 中的初始化过程可以得知，文件中描述了一个如图1.1所示的 cache：

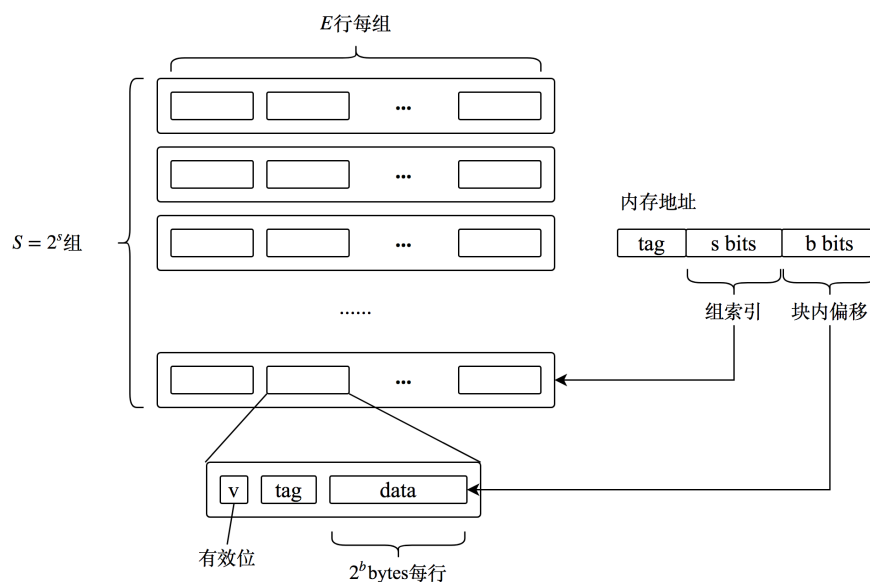


图 1.1: Cache 结构

了解模拟 cache 在内存中的结构后，每一次的内存访问过程如下：首先判断此内存地址是否能够在对应位置 cache hit，如果 cache hit 则将 hit 计数加一，否则判定为 cache miss，将 miss 计数加一，然后判断是否需要替换。如果需要替换则按照 LRU 规则进行替换，然后将 eviction 计数加一。由于 cache line 是以数组的形式存在于内存中的，因此在实现 LRU 的队列结构的时候需要注意对于结构体中 lru 字段的操作。

### 1.2.2 详细设计

接下来根据 LRU 的替换规则设计 accessData 中具体的 cache 替换流程。首先，根据传入的地址计算组索引以及 tag，并根据组索引获得 cache 组。接下来对于组中的所有 cache line 进行遍历，若某一个 cache line 的 tag 字段与事先计算的 tag 字段相等，则增加一次 cache hit 计数，然后更新此 cache set 中所有 cache line 的 lru 字段。若遍历完成都没有 cache hit，则一定出现了 cache miss，此时将 cache miss 计数加一。但此时依然需要分类讨论：若此 cache set 未被填满，则不需要进行替换，直接将新的记录插入 cache set 中并更新所有 cache line 的 lru 字段，否则还需要进行 cache 替换。替换方法为：首先遍历次 cache set 中所有的 cache line，找出其中 lru 值最大的 cache line（存在时间最长的），将其替换为新的地址后更新其他所有 cache line 的 lru 字段。具体的替换流程如图1.2所示。

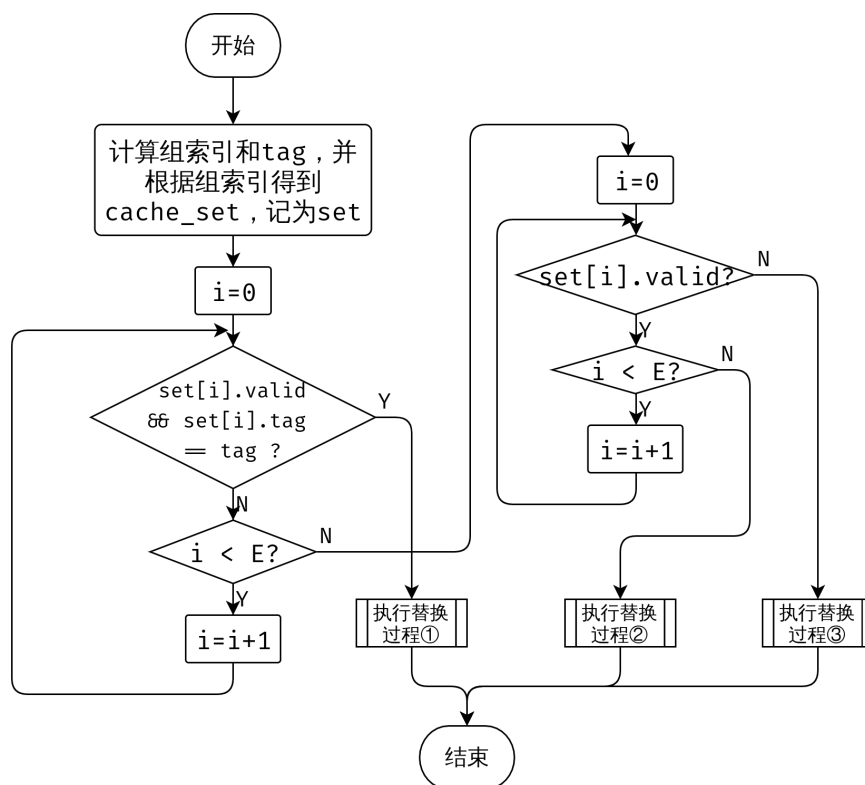


图 1.2: cache 更新流程

在上述三种不同的情况中，三种 lru 的更新方法是不同的：在 cache hit 时，遍历所有 cache line，将 lru 字段小于命中的 cache 的 lru 字段值，并且 valid 字段为 1 的所有 cache line 的 lru 值加一，然后将命中的 cache line 的 lru 值置为 0。

在 cache miss 但未发生 cache eviction 时, 首先遍历所有 cache line, 找出第一个 valid 字为 0 的 cache line, 将这一 cache line 的 valid 字段置为 1, lru 字段置为 0 并将其他所有 valid 字段为 1 的 cache line 的 lru 值加一。

若出现 cache miss eviction, 则遍历所有 cache line, 找出 lru 值最大的 cache line, 将其 lru 值置为 0 并将这一 cache set 中其他所有 cache line 的 lru 值加一。则这三种替换过程如 1.3 所示。

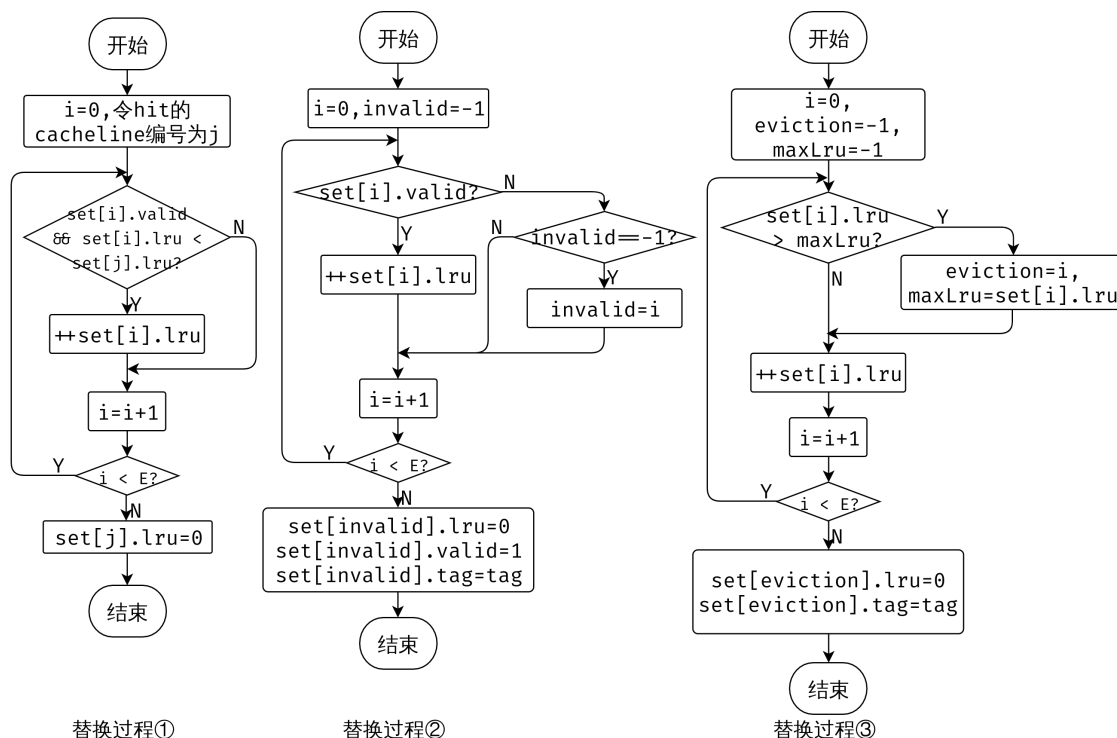


图 1.3: 三种情况下的 cache 替换过程

除了上述对于核心部分的设计之外, 还需要考虑其他的部分: 题目要求实现与 csim-ref 完全相同的功能, 而 csim-ref 的参数可以指定 -h、-v、-s、-E、-b、-t。其中 -h 在所给出的代码框架中已经得以实现, 而 -s、-E、-b、-t 全部是为核心部分的 cache 模拟提供支持的。-v 参数则需要添加额外的实现。使用 -v 参数得到的一个样例输出如下:

```

1 L 10,1 miss
2 M 20,1 miss hit
3 L 22,1 hit
4 S 18,1 hit
5 L 110,1 miss eviction
6 L 210,1 miss eviction
7 M 12,1 miss eviction hit
8 hits:4 misses:5 evictions:3
  
```

从输出中可以看出, -v 对于除了 I 型指令之外的内存操作均进行了输出, 输出格式与 valgrind 的输出格式类似, 但在每行后面添加了 cache hit/miss/eviction 的情况。

## 1.3 实验过程

### 1.3.1 实验环境

表 1.1: 实验环境配置

操作系统	Archlinux x64 2018-04-11 更新
编译器	gcc 7.3.1
Makefile 管理器	gnu make 4.2.1
内存调试工具	valgrind 3.13.0
版本管理工具	git 2.17.0

### 1.3.2 详细步骤

有了上述设计后，代码的实现就较为容易了。首先，补全 freeCache 程序中的代码。根据 initCache 中的代码可知源程序是在 cache 这一二维数组的两个维度进行了分配，而从逻辑上可以推断出，在整个程序的运行过程中不需要对于内存进行新的分配，因此，在 freeCache 时只需要对应的将 initCache 中分配的内存逐个释放即可。实现的代码如下：

```

1 void freeCache() {
2     int i;
3     for (i = 0; i < S; ++i)
4         free(cache[i]);
5     free(cache);
6 }
```

接下来实现核心部分代码，也就是 accessData 模拟 cache 更新这一部分。首先通过如下代码计算组编号以及 tag 值，并根据组编号获得 cache 组。

```

1 mem_addr_t set_index = (addr >> b) & set_index_mask;
2 mem_addr_t tag = addr >> (s + b);
3 cache_set_t cache_set = cache[set_index];
```

然后对于 cache set 进行第一次遍历。在每一次循环中，对于是否命中进行判断，判断的依据为 valid 是否为 1 且 tag 是否与当前内存地址的 tag 相等。若命中则增加命中计数并进行上述替换过程 1，然后直接在返回。此外，还需要注意对于 -v 参数的支持：如果 verbosity flag 为 1，则需要输出“hit”提示。

```

1 for (i = 0; i < E; ++i) {
2     /* hit */
3     if (cache_set[i].valid && cache_set[i].tag == tag) {
4         if (verbosity)
5             printf("hit ");
6         ++hit_count;
```

```

7
8      /* update entry whose lru is less than the current lru (newer) */
9      for (int j = 0; j < E; ++j)
10         if (cache_set[j].valid && cache_set[j].lru < cache_set[i].lru)
11             ++cache_set[j].lru;
12         cache_set[i].lru = 0;
13     return;
14 }
15 }
16 }

```

如果上述循环执行完成而没有返回,表明没有发生 cache hit,因此使用以下代码将 miss 技术加一,并在 verbosity flag 为 1 时打印“miss”提示。

```

1  if (verbosity)
2      printf("miss ");
3  ++miss_count;

```

接下来需要分情况讨论是否会发生 cache eviction。经过考虑后发现,可以将不发生 eviction 情况下寻找最大 lru 条目的过程与发生 eviction 情况下寻找第一个 invalid cache line 的过程合并并在同一个循环中,以提高 cache 模拟器的性能,合并后的循环如下:

```

1  int j, maxIndex = 0;
2  unsigned long long maxLru = 0;
3  for (j = 0; j < E && cache_set[j].valid; ++j) {
4      if (cache_set[j].lru >= maxLru) {
5          maxLru = cache_set[j].lru;
6          maxIndex = j;
7      }
8  }
9  }

```

在上述循环结束后,通过  $j$  的值来判断是否发生了 cache eviction。由于上述循环在遇到 invalid cache line 时会跳出,因此当  $j$  小于  $E$  时表明未发生 cache eviction。也就是说,如果发生 cache eviction,则  $j == E$  一定成立。因此通过以下代码进行 cache 的更新。

```

1  if (j != E) {
2      for (int k = 0; k < E; ++k)
3          if (cache_set[k].valid)
4              ++cache_set[k].lru;
5      cache_set[j].lru = 0;
6      cache_set[j].valid = 1;
7      cache_set[j].tag = tag;
8  } else {
9      if (verbosity)
10         printf("eviction ");
11         ++eviction_count;
12         for (int k = 0; k < E; ++k)

```



```

13         ++cache_set[k].lru;
14         cache_set[maxIndex].lru = 0;
15         cache_set[maxIndex].tag = tag;
16     }

```

在上述代码中,  $j \neq E$  部分为不发生 eviction 的情况,  $j == E$  的部分为发生 eviction 的情况, 分别按照章节1.2.2中的方法进行 cache 的更新。

在 accessData 中核心部分的代码完成后, 需要完成 replayTrace 中的代码调用 accessData 完成对于内存访问过程中 cache 变化的模拟。使用 fscanf 读入每一行, 然后根据访问类型进行对于 accessData 的调用: 忽略 I 型访问, 对于 L 型和 S 型访问调用 accessData 一次, 而对于 M 型访问调用 accessData 两次, 此外还需要注意 verbosity flag 对于输出的影响。代码如下:

```

1  while (fscanf(trace_fp, " %c %llx,%d", buf, &addr, &len) > 0) {
2      if (verbosity && buf[0] != 'I')
3          printf("%c %llx,%d ", buf[0], addr, len);
4      switch (buf[0]) {
5          case 'I':
6              break;
7          case 'L':
8          case 'S':
9              accessData(addr);
10             break;
11          case 'M':
12              accessData(addr);
13              accessData(addr);
14              break;
15          default:
16              break;
17      }
18      if (verbosity && buf[0] != 'I')
19          putchar('\n');
20  }

```

至此, 所有需要填写的代码已补充完成。

### 1.3.3 测试与分析

对于完成的代码进行测试: 首先使用 make 对代码进行编译, 然后直接运行 ./test-csim 命令。实验的测试程序给出的测试样例如表1.2所示, 其输出结果如图1.4所示。

表 1.2: 测试样例

测试文件	组索引位数 s	关联度 E	块偏移位数 b	hit	miss	eviction
yi2.trace	1	1	1	9	8	6
yi.trace	4	2	4	4	5	2

dave.trace	2	1	4	2	3	1
trans.trace	2	1	3	167	71	67
trans.trace	2	2	3	201	37	29
trans.trace	2	4	3	212	26	10
trans.trace	5	1	5	213	7	0
long.trace	5	1	5	265189	21775	21743

```

~/H/CacheLab  dev *... src  ./test-csim  227ms < Fri Apr 13 14:43:34
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
~/H/CacheLab  dev *... src  217ms < Fri Apr 13 14:43:40 2018

```

图 1.4: test-csim 输出结果

从图中可以看出，所有的测试通过。接下来对于 -v 选项进行测试。首先对于 yi.trace 的进行测试，然后对比处理其他 trace 文件的输出与 csim-ref 处理相同文件的输出。运行的结果如1.5所示。

```

[husixu@Ruby src]$ ./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss
M 12,1 hit hit
hits:5 misses:4 evictions:0
[husixu@Ruby src]$ diff <./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace> <./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace>
[husixu@Ruby src]$ diff <./csim -v -s 8 -E 2 -b 4 -t traces/yi2.trace> <./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi2.trace>
[husixu@Ruby src]$ diff <./csim -v -s 8 -E 2 -b 4 -t traces/trans.trace> <./csim-ref -v -s 8 -E 2 -b 4 -t traces/trans.trace>
[husixu@Ruby src]$ diff <./csim -v -s 8 -E 2 -b 4 -t traces/long.trace> <./csim-ref -v -s 8 -E 2 -b 4 -t traces/long.trace>
[husixu@Ruby src]$ diff <./csim -v -s 8 -E 2 -b 4 -t traces/dave.trace> <./csim-ref -v -s 8 -E 2 -b 4 -t traces/dave.trace>
[husixu@Ruby src]$

```

图 1.5: 对于 -v 选项的测试

可以看出，csim 的输出与 csim-ref 的输出完全相同。至此，所有的测试完成，csim 能够完全复现 csim-ref 的功能。

## 第 2 章 Optimizing Matrix Transpose

### 2.1 实验要求

在这一部分的实验中，需要实现 cache 友好的矩阵转置。矩阵转置是这样一种操作：对于一个矩阵  $A$  的转置  $A^T$ ,  $[A^T]_{ij} = [A]_{ji}$ 。在本实验中，将采用 1KB 直接映射，块大小为 32bytes ( $E = 1, C = 1024, B = 32, s = 5, b = 5$ ) 的 cache 配置进行模拟以及测试。矩阵转置的目标是最小化 cache miss 次数。

在实现时，对于矩阵转置程序做出如下限制：

- 显示栈空间的使用：在转置函数中最多使用 12 个 int 类型的局部变量，而不能以任何其他形式使用额外的变量。
- 不能使用递归，如果使用辅助函数，则在任意时刻，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。
- 转置函数不能够改变矩阵 A 但是可以任意操作矩阵 B。
- 不允许在代码中定义任何矩阵以及使用 malloc 及其变种。

测试所使用的数据如下：

- $32 \times 32$  的矩阵，cache miss 次数小于 300 次得 8 分，大于 60 次得 0 分，处于中间的部分按照线性给分。 $64 \times 64$  的矩阵，cache miss 次数小于 1300 次得 8 分，大于 2000 次得 0 分，中间部分按照线性给分。 $62 \times 67$  的矩阵，cache miss 次数小于 2000 次得 10 分，大于 3000 次得 0 分，中间部分按照线性给分。

### 2.2 实验设计

#### 2.2.1 总体设计

在设计 cache 友好的转置之前，首先考虑普通的矩阵转置算法：

```
1 void trans(int M, int N, int A[N][M], int B[M][N]) {  
2     int i, j;  
3     for (i = 0; i < N; ++i)  
4         for (j = 0; j < M; ++j)
```

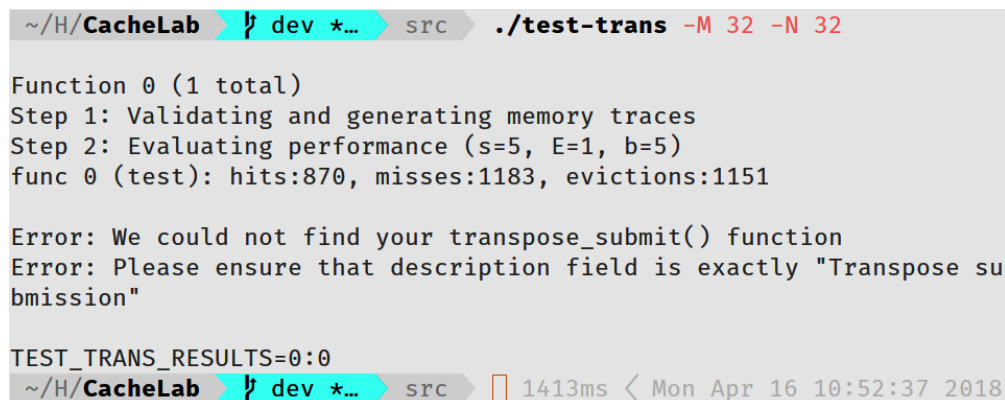
```

5           B[j][i] = A[i][j];
6   }

```

这一函数的正确性毋庸置疑,但它并不是 cache 友好的,因为其并没有有效的利用 cache。以  $32 \times 32$  的矩阵为例,由于 cache 块大小为 32bytes,也就是说一行能够存入 8 个整型变量,此时整个 cache 能够存入矩阵的 8 行。此时,如果同一个矩阵中同一列的两个整数相差了 8 行,那么先后访问这两个整数就会发生 eviction。

在上述平凡算法中,仅是在将矩阵的第一行转置的写入操作中就会发生 24 次 cache eviction,而此后对于每一行的转置对应的 B 中的每一行的写入都会发生 32 次 cache eviction,加上矩阵 A 和矩阵 B 同一行的冲突以及读取矩阵 A 时发生的冲突,一共会发生  $32 + 32 \times 32 + 32 + 24 \times 4 = 1184$  次 cache miss,其中第一个 32 是首次读入矩阵 A 的前八行发生的 cache miss,  $32 \times 32$  是在将 A 的每一行存入 B 的每一列是发生的 cache miss,第二个 32 是存取对角线元素时存入 B 导致 A 先前存入 cache 的部分发生 cache miss,而最后的  $24 \times 4$  则是读取 A 时每读取 4 行就会发生的 24 次 cache miss。实际计算的结果如图 2.1 所示,这与计算的结果非常相近。



```

~/H/CacheLab dev *... src ./test-trans -M 32 -N 32

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test): hits:870, misses:1183, evictions:1151

Error: We could not find your transpose_submit() function
Error: Please ensure that description field is exactly "Transpose submission"

TEST_TRANS_RESULTS=0:0
~/H/CacheLab dev *... src 1413ms < Mon Apr 16 10:52:37 2018

```

图 2.1: 普通转置运算 cache miss 结果

为了减少实际的 cache miss 数,需要尽可能的利用时间局部性和空间局部性:

- 时间局部性: 尽可能的利用当前的 cache block。
- 空间局部性: 在尽可能小的范围内访问数据。

此时,最容易想到的方法是矩阵分块和修改访问次序。就矩阵分块而言,需要考虑分块的大小,以便能够尽量高地利用 cache。由于本实验 cache 的大小是每行 32 字节,因此分块大小应为  $8 \times 8$ ,这样分块的每行就刚好能够存入 cache 的一行,从而有可能提高空间和时间的局部性。

## 2.2.2 详细设计

在  $32 \times 32$  的矩阵中,如果读取或存放的两个数之间相差整数倍,则一定会发生一次 cache miss,但在该矩阵转置的过程中,这对于对角线上的元素是不可能发生的。因此对于非对角线上的 12 个分块,读 + 写的 cache miss 数为  $8 \times 12 + 8 \times 12 = 192$  次,对于对角线上的分块读取 A 发生  $8 \times 4 = 32$  次 cache miss, B 写入则发生  $16 \times 4 = 64$  次 cache miss,总共  $192 + 32 + 64 = 288$  次 cache miss。这样就满足题目要求了。

而对于  $64 \times 64$  的矩阵而言, 由于整个 cache 只能装下矩阵的前 4 行, 经过初步计算后发现, 如果使用与  $32 \times 32$  的矩阵相同的方法, 则对于每个分块转置需要 cache miss 次数为  $8 + 8 \times 8 = 72$  次其中 8 次为读取 A 所发生的 cache miss, 而  $8 \times 8$  次则是由于 cache 只能装下矩阵的 4 行而导致的 B 写入是发生的 cache miss, 于是转置所有分块会发生  $64 \times 72 = 4608$  次 cache miss。实际的初步实验结果为 4611 次, 与理论结果相近, 如图 2.2 所示。

```
~/H/CacheLab dev *... src ./test-trans -M 64 -N 64

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:3586, misses:4611, evictions:4579

Summary for official submission (func 0): correctness=1 misses=4611

TEST_TRANS_RESULTS=1:4611
~/H/CacheLab dev *... src 3637ms < Mon Apr 16 21:57:12 2018
```

图 2.2: 使用  $8 \times 8$  分块转置  $64 \times 64$  矩阵的 cache miss 数

由于主要的 cache miss 发生在存入 B 矩阵的过程中, 而发生如此多的 cache miss 的原因是当矩阵变大后, cache 只能存入矩阵的 4 行, 因此推测如果使用  $4 \times 4$  的分块应该能够减少 cache miss 次数。通过进一步的试验, 发现如果使用  $4 \times 4$  的分块, 则 cache miss 的次数为 1699 次, 如图 2.3 所示。

```
~/H/CacheLab dev *... src ./test-trans -M 64 -N 64

Function 0 (1 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

Summary for official submission (func 0): correctness=1 misses=1699

TEST_TRANS_RESULTS=1:1699
~/H/CacheLab dev *... src 3597ms < Mon Apr 16 22:01:51 2018
```

图 2.3: 使用  $4 \times 4$  分块转置  $64 \times 64$  矩阵的 cache miss 数

虽然已经大大减小了 cache miss 的次数, 然而依然不符合题目要求的 1300 次。其原因应该是使用  $4 \times 4$  的矩阵不能够很好的利用全部的 cache 空间, 每一次存取时 cache line 中一半的元素未能很好的被利用。

为了能够更好的利用空进局部性, 将 B 的一部分作为 cache 使用。依然使用  $8 \times 8$  的分块, 但在写入 B 矩阵时可以在满足 cache 不被替换的情况下首先将 A 的部分元素写入 B 的错误位置, 然后在后续的操作中在考虑将其移至正确的位置。经过计算, 采用如图 2.4 所示的策略进行对分块的转置效率较高。

图 2.4 表示对于 B 的写入过程, 其中的阴影部分为当前步骤更新的元素, 箭头表示更新的顺序。在步骤 1 中, 读取 A 的第 1 行, 并将前后部分分别填入  $B_{11} \sim B_{41}$  以及  $B_{15} \sim B_{45}$  的部分。此时为了减

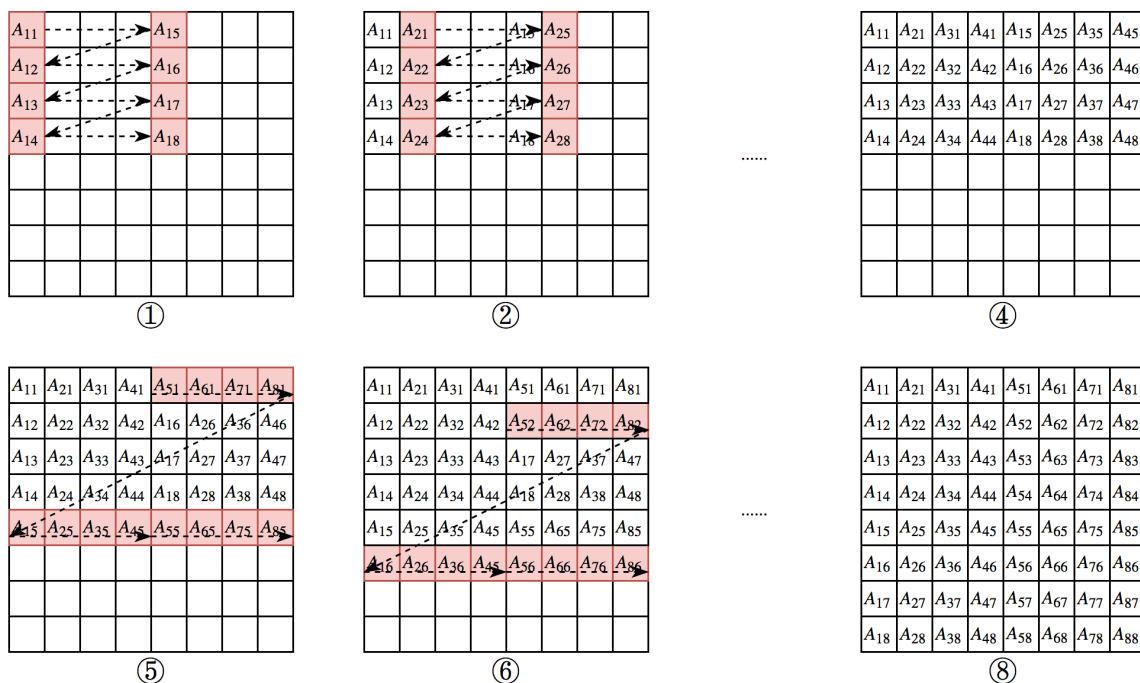


图 2.4: 分块转置策略

少 cache miss, A 第一行的后半部分被填入了错误的地方。接下来依次读入 A 的 2 ~ 4 行并按照上述方法填入 B 对应的位置, 这分别对应图2.4中的步骤 2 ~ 4。

当步骤 1 ~ 4 完成后, 就进行较为复杂也较为重要的步骤。以第 5 步为例, 使用 8 个局部变量中的 4 个存储  $A_{51} \sim A_{81}$ , 并使用另外 4 个局部变量存储第 1 步存储到 B 中的  $A_{15} \sim A_{45}$ 。然后, 首先将  $A_{51} \sim A_{81}$  存储到  $B_{15} \sim B_{18}$  的位置, 接下来将空出来的这 4 个局部变量用于存储  $A_{55} \sim A_{85}$ , 最后将  $A_{15} \sim A_{85}$  存储到  $B_{51} \sim B_{58}$  的位置。这样, 就同时完成了 A 分块中的右半部分的转置以及坐下部分的转置。

使用这样的转置方法, 对于每一个非对角块的转置而言, 步骤 1 ~ 4 中, 除了第一步外, 每一步对 A 的读取发生 1 次 cache miss, 对 B 的写入没有 cache miss, 而第一步对 B 的写入发生 4 次 cache miss, 一共  $4 + 3 \times 1 = 7$  次。在步骤 5 ~ 8 中, 除了第 5 步外, 读取 A 的一列 4 个元素不发生 cache miss, 而第 5 步发生 4 次。读取 B 的一行 4 个以及在同样的位置存入不发生 cache miss。然后再次对于 A 的读取不发生 cache miss (前面的读取已经缓存), 然后对 B 写入 8 个元素发生 1 次 cache miss。因此在步骤 5 ~ 8 中, 一共会发生  $4 + 1 + 3 \times 1 = 8$  次。

对于中心块的转置而言, 在第 1 ~ 4 以及 5 ~ 8 步分别比非中心块多出 4 次和 16 次。因此对于整个矩阵的转置会发生  $56 \times (7 + 8) + 8 \times (7 + 8 + 4 + 16) = 1120$  次。这样, 就能够满足题目要求了。

对于  $61 \times 67$  的矩阵, 由于其条件比较宽松, 预估使用普通分块即可达到要求。这一点将在接下来的章节中继续进行实验验证。

## 2.3 实验过程

### 2.3.1 实验环境

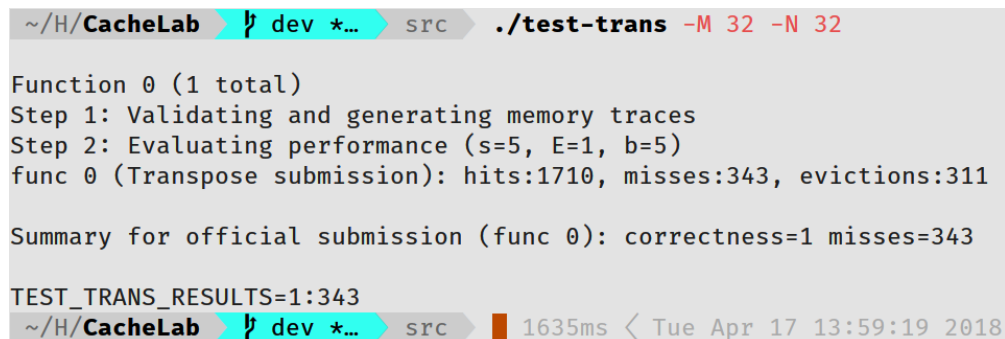
实验环境同上章节1.3.1。

### 2.3.2 详细步骤

首先，处理  $32 \times 32$  的矩阵。使用 4 个变量作为循环变量，进行分块转置后的代码如下：

```
1  for (a = 0; a < N; a += 8)
2      for (b = 0; b < M; b+=8)
3          for(c = a; c < a + 8; ++c)
4              for(d = b; d < b + 8; ++d)
5                  B[d][c] = A[c][d];
```

然而，经过测试后发现，发生了 343 次 cache miss 如图2.5，未能达到题目要求。



```
~/H/CacheLab dev *... src ./test-trans -M 32 -N 32

Function 0 (1 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

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

TEST_TRANS_RESULTS=1:343
~/H/CacheLab dev *... src 1635ms < Tue Apr 17 13:59:19 2018
```

图 2.5: 首次对于  $32 \times 32$  矩阵进行分块转置测试

推测原因，是由于对角先访问的问题：由于 A 和 B 在内存中的位置是连续的，因此 A 和 B 的同一行被映射到了同一个 cache line。此时进行对角线访问时一定会发生 cache eviction。而由于 A 和 B 的元素是一个个处理的，因此会造成多次 cache miss。要解决这一问题，需要一次性读入一整行。好在有 12 个局部变量可以使用，使用 4 个作为循环变量后还有 8 个可以用于存储临时值。修改后的代码如下：

```
1  for (a = 0; a < N; a += 8) {
2      for (b = 0; b < M; b += 8) {
3          for (c = a; c < a + 8; ++c) {
4              reg0 = A[c][b + 0]; reg1 = A[c][b + 1];
5              reg2 = A[c][b + 2]; reg3 = A[c][b + 3];
6              reg4 = A[c][b + 4]; reg5 = A[c][b + 5];
7              reg6 = A[c][b + 6]; reg7 = A[c][b + 7];
8              B[b + 0][c] = reg0; B[b + 1][c] = reg1;
9              B[b + 2][c] = reg2; B[b + 3][c] = reg3;
10             B[b + 4][c] = reg4; B[b + 5][c] = reg5;
11             B[b + 6][c] = reg6; B[b + 7][c] = reg7;
```

```

12     }
13     }
14 }

```

对于上述代码重新进行编译并测试，结果如图2.6所示，共 287 次 cache miss，达到题目要求，且与计算结果几乎一致。

```

~/H/CacheLab dev *... src ./test-trans -M 32 -N 32

Function 0 (1 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

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

TEST_TRANS_RESULTS=1:287
~/H/CacheLab dev *... src 1860ms < Tue Apr 17 14:06:08 2018

```

图 2.6: 对于修改后的代码进行测试

接下来按照详细设计中的转置策略对于  $64 \times 64$  的代码进行矩阵转置测试：

```

1  for (a = 0; a < N; a += 8) {
2      for (b = 0; b < M; b += 8) {
3          for (c = a; c < a + 4; ++c) {
4              reg0 = A[c][b + 0]; reg1 = A[c][b + 1];
5              reg2 = A[c][b + 2]; reg3 = A[c][b + 3];
6              reg4 = A[c][b + 4]; reg5 = A[c][b + 5];
7              reg6 = A[c][b + 6]; reg7 = A[c][b + 7];
8              B[b + 0][c] = reg0; B[b + 0][c + 4] = reg4;
9              B[b + 1][c] = reg1; B[b + 1][c + 4] = reg5;
10             B[b + 2][c] = reg2; B[b + 2][c + 4] = reg6;
11             B[b + 3][c] = reg3; B[b + 3][c + 4] = reg7;
12         }
13         for (c = 0; c < 4; ++c) {
14             reg0 = A[a + 4][b + c]; reg1 = A[a + 5][b + c];
15             reg2 = A[a + 6][b + c]; reg3 = A[a + 7][b + c];
16             reg4 = B[b + c][a + 4]; reg5 = B[b + c][a + 5];
17             reg6 = B[b + c][a + 6]; reg7 = B[b + c][a + 7];
18             B[b + c][a + 4] = reg0; B[b + c][a + 5] = reg1;
19             B[b + c][a + 6] = reg2; B[b + c][a + 7] = reg3;
20             reg0 = A[a + 4][b + c + 4]; reg1 = A[a + 5][b + c + 4];
21             reg2 = A[a + 6][b + c + 4]; reg3 = A[a + 7][b + c + 4];
22             B[b + c + 4][a + 0] = reg4; B[b + c + 4][a + 4] = reg0;
23             B[b + c + 4][a + 1] = reg5; B[b + c + 4][a + 5] = reg1;
24             B[b + c + 4][a + 2] = reg6; B[b + c + 4][a + 6] = reg2;
25             B[b + c + 4][a + 3] = reg7; B[b + c + 4][a + 7] = reg3;
26         }

```



```

27     }
28 }

```

上述代码中，**for** ( $c = a; c < a + 4; ++c$ ) 循环内执行的代码对应图 2.4 策略中的第 1 ~ 4 步，而 **for** ( $c = 4; c < 4; ++c$ ) 中执行的代码则对应图 2.4 策略中的第 5 ~ 8 步。

下面对于上述代码进行测试。运行 Make 进行编译后，执行 `./test-trans -M 64 -N 64` 进行测试，结果如图 2.7 所示。

```

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9106, misses:1139, evictions:1107
Summary for official submission (func 0): correctness=1 misses=1139
TEST_TRANS_RESULTS=1:1139
~/H/CacheLab dev *... src 5.1s < Tue Apr 17 14:17:09 2018

```

图 2.7:  $64 \times 64$  矩阵分块转置测试

从图中可以看出，cache miss 的次数为 1139 次，满足题目要求。

对于  $61 \times 67$  的矩阵，其策略也是使用分块来优化 cache 读写。然而 61 和 67 均为质数，无法找到一个明显的规律看出多少间隔能够填满一个 cache。但由于要求比较宽松，因此无需考虑对角线处理，直接使用普通分块操作进行尝试。最后发现， $23 \times 23$  的分块能够满足要求，其代码如下：

```

1  for (a = 0; a < N; a += 23)
2      for (b = 0; b < M; b += 23)
3          for (c = a; c < a + 23 && c < N; ++c)
4              for (d = b; d < b + 23 && d < M; ++d)
5                  B[d][c] = A[c][d];

```

测试的结果如图 2.8 所示，共发生了 1928 次 cache miss，小于要求的 2000 次。

```

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6251, misses:1928, evictions:1896
Summary for official submission (func 0): correctness=1 misses=1928
TEST_TRANS_RESULTS=1:1928
~/H/CacheLab dev *... src 5.3s < Tue Apr 17 14:27:00 2018

```

图 2.8:  $61 \times 67$  矩阵分块转置测试

### 2.3.3 测试与分析

在完成了上述代码后，将所有代码整合进 `transpose_submit` 函数中，然后在 `registerFunctions` 函数中注册 `transpose_submit` 函数。整合后的函数如下：

```

1 void transpose_submit(int M, int N, int A[N][M], int B[M][N]) {
2     int a, b, c, d;
3     int reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7;
4     if (M == 32 && N == 32) {
5         ...
6         /* 32 * 32 matrix transpose code */
7     } else if (M == 64 && N == 64) {
8         ...
9         /* 64 * 64 matrix transpose code */
10    } else if (M == 61 && N == 67) {
11        ...
12        /* 61 * 67 matrix transpose code */
13    }
14 }

```

编译完成后，运行 `python2 driver.py` 进行测试，结果如图2.9所示，从图中可以看出，所有测试均通过，并达到题目要求。至此，所有的实验代码编写以及测试完成。

```

~/H/CacheLab  dev *...  src  python2 driver.py  Tue Apr 17 14:39:21 2018
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	287
Trans perf 64x64	8.0	8	1139
Trans perf 61x67	10.0	10	1928
Total points	53.0	53	

```

~/H/CacheLab  dev *...  src  12s < Tue Apr 17 14:39:34 2018

```

图 2.9: 最终测试结果

# 心得体会

通过这次实验，我对于 cache 的结构，运作方式以及如何更高效的写出 cache friendly 的代码有了更为深入的了解。通过对于 cache 的模拟，我对于 cache 的组成方式，与内存地址的对应关系以及 cache 组相连的工作方式有了更为深入的认识。

在对于错误代码的调试过程中，我尝试使用了各种方法来结构性的显示 cache 的内容以及替换的过程，这也给了我有关 cache 工作方式更为直观的印象。

第二个实验则更令我印象深刻，它从简单的矩阵变换入手，展示了对于程序员而言透明的 cache 是如何影响代码性能的，以及程序员应该如何优化代码使之能够在 cache 层面有更好的工作效率。通过逐步深入的实验，对于 cache 的理解也逐步加深，并在实验查阅资料的过程中了解到了 cache 优化在包括科学计算等多个领域的应用，这不仅开拓了我的视野，也为今后的学习工作奠定了基础。

# 源代码

csim.c

```
1  /*
2   * csim.c - A cache simulator that can replay traces from Valgrind
3   *          and output statistics such as number of hits, misses, and
4   *          evictions. The replacement policy is LRU.
5   *
6   * Implementation and assumptions:
7   * 1. Each load/store can cause at most one cache miss. (I examined the trace,
8   * the largest request I saw was for 8 bytes).
9   * 2. Instruction loads (I) are ignored, since we are interested in evaluating
10  * trans.c in terms of its data cache performance.
11  * 3. data modify (M) is treated as a load followed by a store to the same
12  * address. Hence, an M operation can result in two cache hits, or a miss and
13  * a
14  * hit plus an possible eviction.
```

```

14  *
15  * The function printSummary() is given to print output.
16  * Please use this function to print the number of hits, misses and evictions.
17  * This is crucial for the driver to evaluate your work.
18  */
19  #include <getopt.h>
20  #include <stdlib.h>
21  #include <unistd.h>
22  #include <stdio.h>
23  #include <assert.h>
24  #include <math.h>
25  #include <limits.h>
26  #include <string.h>
27  #include <errno.h>
28  #include "cachelab.h"
29
30  // #define DEBUG_ON
31  #define ADDRESS_LENGTH 64
32
33  /* Type: Memory address */
34  typedef unsigned long long int mem_addr_t;
35
36  /* Type: Cache line
37     LRU is a counter used to implement LRU replacement policy */
38  typedef struct cache_line {
39     char valid;
40     mem_addr_t tag;
41     unsigned long long int lru;
42 } cache_line_t;
43
44  typedef cache_line_t *cache_set_t;
45  typedef cache_set_t *cache_t;
46
47  /* Globals set by command line args */
48  int verbosity = 0; /* print trace if set */
49  int s = 0; /* set index bits */
50  int b = 0; /* block offset bits */
51  int E = 0; /* associativity */
52  char *trace_file = NULL;
53
54  /* Derived from command line args */
55  int S; /* number of sets */
56  int B; /* block size (bytes) */
57
58  /* Counters used to record cache statistics */
59  int miss_count = 0;

```

```

60 int hit_count = 0;
61 int eviction_count = 0;
62 unsigned long long int lru_counter = 1;
63
64 /* The cache we are simulating */
65 cache_t cache;
66 mem_addr_t set_index_mask;
67
68 /*
69  * initCache - Allocate memory, write 0's for valid and tag and LRU
70  * also computes the set_index_mask
71  */
72 void initCache() {
73     int i, j;
74     cache = (cache_set_t *) malloc(sizeof(cache_set_t) * S);
75     for (i = 0; i < S; i++) {
76         cache[i] = (cache_line_t *) malloc(sizeof(cache_line_t) * E);
77         for (j = 0; j < E; j++) {
78             cache[i][j].valid = 0;
79             cache[i][j].tag = 0;
80             cache[i][j].lru = 0;
81         }
82     }
83
84     /* Computes set index mask */
85     set_index_mask = (mem_addr_t) (pow(2, s) - 1);
86 }
87
88
89 /*
90  * freeCache - free allocated memory
91  */
92 void freeCache() {
93     int i;
94     for (i = 0; i < S; ++i)
95         free(cache[i]);
96     free(cache);
97 }
98
99
100 /*
101  * accessData - Access data at memory address addr.
102  *   If it is already in cache, increast hit_count
103  *   If it is not in cache, bring it in cache, increase miss count.
104  *   Also increase eviction_count if a line is evicted.
105  */

```

```

106 void accessData(mem_addr_t addr) {
107     int i;
108     /* unsigned long long int eviction_lru = ULONG_MAX; */
109     /* unsigned int eviction_line = 0; */
110     mem_addr_t set_index = (addr >> b) & set_index_mask;
111     mem_addr_t tag = addr >> (s + b);
112
113     cache_set_t cache_set = cache[set_index];
114
115     /* judge if hit */
116     for (i = 0; i < E; ++i) {
117         /* hit */
118         if (cache_set[i].valid && cache_set[i].tag == tag) {
119             if (verbosity)
120                 printf("hit ");
121             ++hit_count;
122
123             /* update entry whose lru is less than the current lru (newer) */
124             for (int j = 0; j < E; ++j)
125                 if (cache_set[j].valid && cache_set[j].lru < cache_set[i].lru)
126                     ++cache_set[j].lru;
127             cache_set[i].lru = 0;
128             return;
129         }
130     }
131
132     /* missed */
133     if (verbosity)
134         printf("miss ");
135     ++miss_count;
136
137     /* find the biggest lru or the first invlaid line */
138     int j, maxIndex = 0;
139     unsigned long long maxLru = 0;
140     for (j = 0; j < E && cache_set[j].valid; ++j) {
141         if (cache_set[j].lru >= maxLru) {
142             maxLru = cache_set[j].lru;
143             maxIndex = j;
144         }
145     }
146
147     if (j != E) {
148         /* found an invalid entry */
149         /* update other entries */
150         for (int k = 0; k < E; ++k)
151             if (cache_set[k].valid)

```

```

152         ++cache_set[k].lru;
153     /* insert line */
154     cache_set[j].lru = 0;
155     cache_set[j].valid = 1;
156     cache_set[j].tag = tag;
157 } else {
158     /* all entry is valid, replace the oldest one*/
159     if (verbosity)
160         printf("eviction ");
161     ++eviction_count;
162     for (int k = 0; k < E; ++k)
163         ++cache_set[k].lru;
164     cache_set[maxIndex].lru = 0;
165     cache_set[maxIndex].tag = tag;
166 }
167 }
168
169
170 /*
171  * replayTrace - replays the given trace file against the cache
172  */
173 void replayTrace(char *trace_fn) {
174     char buf[1000];
175     mem_addr_t addr = 0;
176     unsigned int len = 0;
177     FILE *trace_fp = fopen(trace_fn, "r");
178
179     int testcount = 0;
180
181     while (fscanf(trace_fp, " %c %llx,%d", buf, &addr, &len) > 0) {
182         if (verbosity && buf[0] != 'I')
183             printf("%c %llx,%d ", buf[0], addr, len);
184         switch (buf[0]) {
185             case 'I':
186                 break;
187             case 'L':
188             case 'S':
189                 accessData(addr);
190                 ++testcount;
191                 break;
192             case 'M':
193                 accessData(addr);
194                 accessData(addr);
195                 ++testcount;
196                 break;
197             default:

```

```

198             break;
199         }
200         if (verbosity && buf[0] != 'I')
201             putchar('\n');
202     }
203     fclose(trace_fp);
204 }
205
206 void debug() {
207     printf("=====\n");
208     for (int i = 0; i < S; ++i) {
209         for (int j = 0; j < E; ++j)
210             printf("%d,%10llx,%4llu | ", cache[i][j].valid, cache[i][j].tag,
211                 cache[i][j].lru);
212         printf("\n");
213     }
214 }
215 /*
216  * printUsage - Print usage info
217  */
218 void printUsage(char *argv[]) {
219     printf("Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n", argv[0]);
220     printf("Options:\n");
221     printf("  -h          Print this help message.\n");
222     printf("  -v          Optional verbose flag.\n");
223     printf("  -s <num>    Number of set index bits.\n");
224     printf("  -E <num>    Number of lines per set.\n");
225     printf("  -b <num>    Number of block offset bits.\n");
226     printf("  -t <file>   Trace file.\n");
227     printf("\nExamples:\n");
228     printf("  linux> %s -s 4 -E 1 -b 4 -t traces/yi.trace\n", argv[0]);
229     printf("  linux> %s -v -s 8 -E 2 -b 4 -t traces/yi.trace\n", argv[0]);
230     exit(0);
231 }
232
233 /*
234  * main - Main routine
235  */
236 int main(int argc, char *argv[]) {
237     char c;
238
239     while ( (c = getopt(argc, argv, "s:E:b:t:vh")) != -1) {
240         switch (c) {
241             case 's':
242                 s = atoi(optarg);

```



```

243         break;
244     case 'E':
245         E = atoi(optarg);
246         break;
247     case 'b':
248         b = atoi(optarg);
249         break;
250     case 't':
251         trace_file = optarg;
252         break;
253     case 'v':
254         verbosity = 1;
255         break;
256     case 'h':
257         printUsage(argv);
258         exit(0);
259     default:
260         printUsage(argv);
261         exit(1);
262     }
263 }
264
265 /* Make sure that all required command line args were specified */
266 if (s == 0 || E == 0 || b == 0 || trace_file == NULL) {
267     printf("%s: Missing required command line argument\n", argv[0]);
268     printUsage(argv);
269     exit(1);
270 }
271
272 /* Compute S, E and B from command line args */
273 S = (unsigned int) pow(2, s);
274 B = (unsigned int) pow(2, b);
275
276 /* Initialize cache */
277 initCache();
278
279 #ifdef DEBUG_ON
280     printf("DEBUG: S:%u E:%u B:%u trace:%s\n", S, E, B, trace_file);
281     printf("DEBUG: set_index_mask: %llu\n", set_index_mask);
282 #endif
283
284 replayTrace(trace_file);
285
286 /* Free allocated memory */
287 freeCache();
288

```

```

289     /* Output the hit and miss statistics for the autograder */
290     printSummary(hit_count, miss_count, eviction_count);
291     return 0;
292 }

```

### trans.c

```

1  /*
2  * trans.c - Matrix transpose B = A^T
3  *
4  * Each transpose function must have a prototype of the form:
5  * void trans(int M, int N, int A[N][M], int B[M][N]);
6  *
7  * A transpose function is evaluated by counting the number of misses
8  * on a 1KB direct mapped cache with a block size of 32 bytes.
9  */
10 #include <stdio.h>
11 #include "cachelab.h"
12
13 int is_transpose(int M, int N, int A[N][M], int B[M][N]);
14
15 /*
16 * transpose_submit - This is the solution transpose function that you
17 * will be graded on for Part B of the assignment. Do not change
18 * the description string "Transpose submission", as the driver
19 * searches for that string to identify the transpose function to
20 * be graded.
21 */
22 char transpose_submit_desc[] = "Transpose submission";
23 void transpose_submit(int M, int N, int A[N][M], int B[M][N]) {
24     int a, b, c, d;
25     int reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7;
26     if (M == 32 && N == 32) {
27         for (a = 0; a < N; a += 8) {
28             for (b = 0; b < M; b += 8) {
29                 for (c = a; c < a + 8; ++c) {
30                     reg0 = A[c][b + 0];
31                     reg1 = A[c][b + 1];
32                     reg2 = A[c][b + 2];
33                     reg3 = A[c][b + 3];
34                     reg4 = A[c][b + 4];
35                     reg5 = A[c][b + 5];
36                     reg6 = A[c][b + 6];
37                     reg7 = A[c][b + 7];
38                     B[b + 0][c] = reg0;
39                     B[b + 1][c] = reg1;
40                     B[b + 2][c] = reg2;

```

```

41         B[b + 3][c] = reg3;
42         B[b + 4][c] = reg4;
43         B[b + 5][c] = reg5;
44         B[b + 6][c] = reg6;
45         B[b + 7][c] = reg7;
46     }
47 }
48 }
49 } else if (M == 64 && N == 64) {
50     for (a = 0; a < N; a += 8) {
51         for (b = 0; b < M; b += 8) {
52             for (c = a; c < a + 4; ++c) {
53                 reg0 = A[c][b + 0];
54                 reg1 = A[c][b + 1];
55                 reg2 = A[c][b + 2];
56                 reg3 = A[c][b + 3];
57                 reg4 = A[c][b + 4];
58                 reg5 = A[c][b + 5];
59                 reg6 = A[c][b + 6];
60                 reg7 = A[c][b + 7];
61                 B[b + 0][c] = reg0;
62                 B[b + 0][c + 4] = reg4;
63                 B[b + 1][c] = reg1;
64                 B[b + 1][c + 4] = reg5;
65                 B[b + 2][c] = reg2;
66                 B[b + 2][c + 4] = reg6;
67                 B[b + 3][c] = reg3;
68                 B[b + 3][c + 4] = reg7;
69             }
70             for (c = 0; c < 4; ++c) {
71                 reg0 = A[a + 4][b + c];
72                 reg1 = A[a + 5][b + c];
73                 reg2 = A[a + 6][b + c];
74                 reg3 = A[a + 7][b + c];
75                 reg4 = B[b + c][a + 4];
76                 reg5 = B[b + c][a + 5];
77                 reg6 = B[b + c][a + 6];
78                 reg7 = B[b + c][a + 7];
79                 B[b + c][a + 4] = reg0;
80                 B[b + c][a + 5] = reg1;
81                 B[b + c][a + 6] = reg2;
82                 B[b + c][a + 7] = reg3;
83                 reg0 = A[a + 4][b + c + 4];
84                 reg1 = A[a + 5][b + c + 4];
85                 reg2 = A[a + 6][b + c + 4];
86                 reg3 = A[a + 7][b + c + 4];

```

```

87         B[b + c + 4][a + 0] = reg4;
88         B[b + c + 4][a + 4] = reg0;
89         B[b + c + 4][a + 1] = reg5;
90         B[b + c + 4][a + 5] = reg1;
91         B[b + c + 4][a + 2] = reg6;
92         B[b + c + 4][a + 6] = reg2;
93         B[b + c + 4][a + 3] = reg7;
94         B[b + c + 4][a + 7] = reg3;
95     }
96 }
97 }
98 } else if (M == 61 && N == 67) {
99     for (a = 0; a < N; a += 23)
100         for (b = 0; b < M; b += 23)
101             for (c = a; c < a + 23 && c < N; ++c)
102                 for (d = b; d < b + 23 && d < M; ++d)
103                     B[d][c] = A[c][d];
104 }
105
106 }
107
108 /*
109  * You can define additional transpose functions below. We've defined
110  * a simple one below to help you get started.
111  */
112
113 /*
114  * registerFunctions - This function registers your transpose
115  * functions with the driver. At runtime, the driver will
116  * evaluate each of the registered functions and summarize their
117  * performance. This is a handy way to experiment with different
118  * transpose strategies.
119  */
120 void registerFunctions() {
121     /* Register your solution function */
122     registerTransFunction(transpose_submit, transpose_submit_desc);
123 }
124
125 /*
126  * is_transpose - This helper function checks if B is the transpose of
127  * A. You can check the correctness of your transpose by calling
128  * it before returning from the transpose function.
129  */
130 int is_transpose(int M, int N, int A[N][M], int B[M][N]) {
131     int i, j;
132

```

```
133     for (i = 0; i < N; i++) {  
134         for (j = 0; j < M; ++j) {  
135             if (A[i][j] != B[j][i]) {  
136                 return 0;  
137             }  
138         }  
139     }  
140     return 1;  
141 }
```