华中科技大学

计算机系统结构实验报告

姓　　名：　　方子豪

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2008

学　　号：　　U202215628

分数	
教师签名	

2025 年　　5 月　 11 日

目录

1. Cache 模拟器实验 3

1.1. 实验目的 3
1.2. 实验环境 3
1.3. 实验思路 4
1.3.2 Cache 的初始化及销毁 4
1.3.2 Cache 的访问，更新和替换 5
1.3.2 Cache 的命令行解析 6
1.4. 实验结果和分析 7

2. 优化矩阵转置实验 8

2.1. 实验目的 8
2.2. 实验环境 8
2.3. 实验思路 8
2.3.1 32*32 矩阵转置 8
2.3.2 64*64 矩阵转置 9
2.3.3 61*67 矩阵转置 10
2.4. 实验结果和分析 11

3. 总结和体会 12

4. 对实验课程的建议 13

1. Cache 模拟器实验

1.1. 实验目的

本实验要求在 csim.c 提供的程序框架中，编写实现一个 Cache 模拟器。模拟器需要根据输入的内存访问轨迹，模拟混存相对内存访问轨迹的命中（hit）/缺失行为（miss），同时能够输出命中、缺失和（缓存行）脱胎/驱除（基于 LRU 算法）的总数。
其中对于 Cache 模拟器的要求为：

（1）实现的 Cache 模拟器应为多路组相连，用参数 s 代表组的二进制位数，参数 B 代表缓存块的位数，参数 E 代表每一组中缓存行的数目。能够根据不同的参数 s，B，E 实现具体的 Cache 模拟器

（2）模拟器能够处理一系列命令：***Usage: ./csim [-hv] -s <s> -E <E> -b <b> -t <tracefile>*** 。其中 s，e，b 为输入参数，t 为数据文件路径，h 为输出帮助信息，v 为输出详细运行过程。

（3）Cache 模拟器能够模拟 LUR 替换算法

1.2. 实验环境

表 1-1 远程实验环境

硬件环境	CPU 架构	X86_64
	CPU 型号	Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz
	CPU 核数	16 核
软件环境	Gcc 版本	Gcc version 7.3.0 (GCC)
	python 版本	Python 2.7.13
	操作系统	Linux educoder 4.19.1-1.el7.elrepo.x86_64

表 1-2 本地实验环境

硬件环境	CPU 架构	X86_64
	CPU 型号	12th Gen Intel（R）Core（TM）i7-12700H
	CPU 核数	14 核
软件环境	Gcc 版本	Gcc version 7.3.0 (GCC)
	python 版本	Python 3.12.0
	操作系统	Linux 5.15.167.4-microsoft-standard-WSL2

1.3. 实验思路

基于 1.1 实验目的中的要求，我们将实验的具体实现分为三个步骤：实现 Cache **的数据结构模拟**，实现 Cache **的初始化**，实现 Cache **的访问、更新和替换策略**，实现**命令行解析。**下面我将分别分析上述四个步骤的具体实现策略以及代码。

1.3.1 Cache 的数据结构模拟

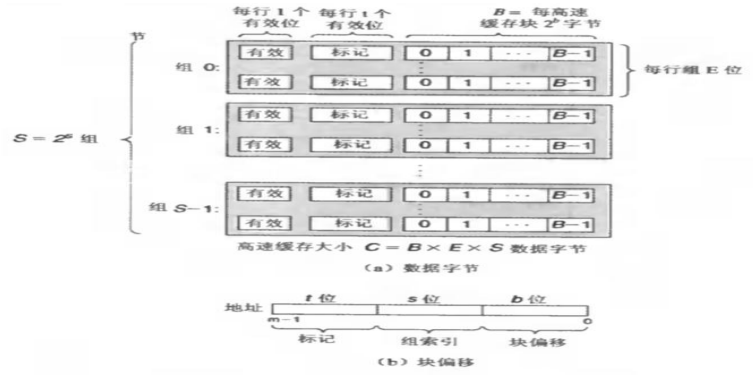

图 1-1 Cache 结构图示

我们需要模拟的 Cache 的具体结构如上图 1-1 所示，可以看到，一个 Cache 可以分为多个 Cache 组，同时一个组又可以分为多个行。因此在设计时需要分开设计。

在具体的实现中，我选择将 Cache 的一个行设置为一个结构体 ***CacheLine***，其中包含了 valid，tag 和 timestamp 的信息，分别代表**有效位**，**标记位**和**时间戳**的信息。由于在本实验中 Cache 具体的存储块不需要我们实现，所以为了简化代码，我没有将其设计在该结构体中。

在设计完 ***CacheLine*** 结构体后还需要设计整体的 CacheSim。在这里我选择使用 ***num_sets***，***num_lines***，***block_size*** 和 ***sets*** 来分别表示 Cache 的组数，Cache 每一组的行数，Cache 的块大小以及一系列指针。其中 ***sets*** 指针是一个二级指针类似于一个二维数组，第一维对应于 Cache 的组数，第二维对应于 Cache 每一组的行数。这样我可以通过 CacheSim 结构体将 CacheLine 串联起来模拟一个真正的 Cache。

```
typedef struct {
 int valid;
 int tag;
 int timestamp;
} CacheLine;

typedef struct {
 int num_sets;
 int num_lines;
 int block_size;
 CacheLine **sets;
} CacheSim;
```

图 1-2 Cache 结构设计

1.3.2 Cache 的初始化及销毁

我们有 s，E，b 三个初始参数，基于这三个参数对我们上述的 Cache 结构进行初始化，其实也就是对指针所指向的理论结构进行物理地址的赋予。第一步就是计算出组数 S = 1 << s，块偏移 B = 1 << b。紧接着对第一维的二级指针进行 malloc 操作。成功之后对第二维的指针进行 malloc 操作同时初始化对应的 CacheLine 的值即可。

```
sim->sets = (CacheLine **)malloc(S * sizeof(CacheLine *));
 for (int i = 0; i < S; ++i) {
  sim->sets[i] = (CacheLine *)malloc(E * sizeof(CacheLine));
  for (int j = 0; j < E; ++j) {
   sim->sets[i][j].valid = 0;
   sim->sets[i][j].tag = -1;
   sim->sets[i][j].timestamp = 0;
  }
 }
```

图 1-3 Cache 的初始化 malloc

当运行程序结束之后，为了程序安全性考虑我们需要释放掉人为分配的内存，对应的操作就是 Cache 的销毁操作。

```
void free_cache() {
 for (int i = 0; i < sim->num_sets; ++i)
  free(sim->sets[i]);
 free(sim->sets);
 free(sim);
}
```

图 1-4 Cache 的销毁

1.3.3 Cache 的访问，更新和替换

本实验中我们需要实现三类操作，**L（数据加载）、S（数据存储）、M（数据存储之后的数据加载）**。由于我们的模拟不涉及 Cache 中块的内容写入等操作，所以对于我们的 L 加载和 S 存储操作来说，对于 Cache 的操作是相同的，即都是访问**一次** Cache 中的对应块。而对于我们的 M 操作即相当于访问**两次** Cache 中的块。如果找到了对应的块则更新 LRU，如果没找到对应的块则替换掉 LRU 最大的值并且更新所有的 LRU。

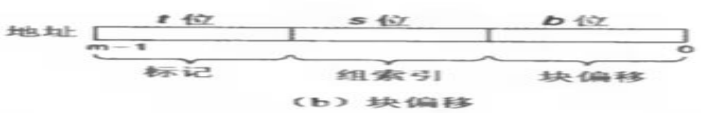

图 1-5 地址对应的信息

通过上图 1-4 我们可以看到一个地址访问可以拆分为三个信息，分别是标记位信息 ***tag***，组索引信息 ***set_index*** 和块偏移。因此得到访问 Cache 组号 ***set_index*** 的信息就非常简单了，即取出地址中的第 b 到 b+s-1 位，实际的操作即为 addr >> b & ((1<<s)-1)。同时利用 addr >> (b+s)可以得到我们的 ***tag*** 信息用于判断在对应的组中是否存在我们需要的块。如果没有则需要调入新块并且将时间戳最大的块给 kick，反之直接更新时间戳即可。

```
void access_cache(unsigned int address, int s_bits, int b_bits) {
 int set_index = (address >> b_bits) & ((1 << s_bits) - 1);
 int tag = address >> (s_bits + b_bits);
 int line = find_hit(set_index, tag);

 if (line != -1) {
  hit_count++;
  if (verbose_flag) printf(" hit");
  update_line(set_index, line, tag);
 } else {
  miss_count++;
  if (verbose_flag) printf(" miss");
  int empty = find_empty_line(set_index);
  if (empty != -1) {
   update_line(set_index, empty, tag);
  } else {
   eviction_count++;
   if (verbose_flag) printf(" eviction");
   int lru = get_lru_line(set_index);
   update_line(set_index, lru, tag);
  }
 }
}
```

图 1-6 根据对应地址求出相应信息访问 Cache 中对应行

其中具体的实现分别有 ***find_hit()***，***find_empty_line()***，***get_lru_line()***，***update_line()***四个函数，功能分别为查找编号为 set_index 的组中是否有标记位为 tag 的行、查找对应编号的组中是否有空行、求出当前编号的组中 LUR 最大的那一行以及更新这一组中所有的时间戳。这四个函数的具体实现都较为简单，基本上都是遍历一遍对应编号为 set_index 的组中的所有行即可实现区别仅在于返回的条件不同以及修改的值不同而已，因此在这里不过多赘述仅给出 ***find_hit()***函数来作为例子。

```
int find_hit(int set_index, int tag) {
 for (int i = 0; i < sim->num_lines; ++i)
  if (sim->sets[set_index][i].valid && sim->sets[set_index][i].tag == tag)
   return i;
 return -1;
}
```

图 1-7 find_hit 函数作为例子，上述四个函数方法均类似该函数

1.3.4 Cache 的命令行解析

我们使用 C 语言中的 getopt.h 库提供的 getopt()函数来读入命令行中的参数。getopt()函数接收 main()函数的 argc 和 argv 以及一个表示命令选项的字符串。其中该函数可以多次调用，每次调用返回一个编译选项，编译选项的值存放在 optarg 中，当读取结束时函数返回-1。因此可以通过该函数来依次实现组索引位数 s，每一组包含的函数 E，位偏移宽度 b 等值的赋值以及文件路径 t 的载入。

```
while ((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1) {
    switch (opt) {
    case 'h':
     printf("help");
     exit(0);
    case 'v':
     verbose_flag = 1;
     break;
    case 's':
     s = atoi(optarg);
     break;
    case 'E':
     E = atoi(optarg);
     break;
    case 'b':
     b = atoi(optarg);
     break;
    case 't':
     strncpy(trace_path, optarg, sizeof(trace_path) - 1);
     break;
    default:
     printf("something went wrong");
     exit(1);
    }
   }
```

图 1-8 命令行解析函数

1.4. 实验结果和分析

完成上述步骤之后在本地编译并运行 Cache 模拟器最终结果成功，提交至头歌上运行测试通过，结果如下：

```
测试模拟器
运行 ./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

yi2.trace测试通过，
yi1.trace测试通过，
dave.trace测试通过，
trace.trace测试通过，
long.trace测试通过!
```

图 1-9 头歌测试通过

2. 优化矩阵转置实验

2.1. 实验目的

本实验要求在 trans.c 中使用 C 语言编写一个实现矩阵转置的函数 transpose_submit。要求尽量使函数调用过程中 cache 不命中数 miss 尽可能少。

同时要求限制对栈的使用，在转置函数中最多定义和使用 12 个 int 类型的变量，不能使用 long 类型的变量或其他位模式数据。

2.2. 实验环境

本实验本地和头歌均可运行，我仅在头歌上验证了该实验，实验环境如下：

表 2-1 头歌实验环境

硬件环境	CPU 架构	X86_64
	CPU 型号	Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz
	CPU 核数	16 核
软件环境	Gcc 版本	Gcc version 7.3.0 (GCC)
	python 版本	Python 2.7.13
	操作系统	Linux educoder 4.19.1-1.el7.elrepo.x86_64

2.3. 实验思路

首先需要分析 Cache 块的大小和容量再来进行分析。由于 ***s=5，E = 1，b = 5***，故可知缓存有 32 行，每组一行，每行存 8 个 int 类型数据，则可知我们**连续八个 int 类型的数据为一组**。

2.3.1 32*32 矩阵转置

首先考虑最简单的 32*32 情况：由于每一行有 4*8 个 int 类型，所以相当于每一行中的数据可以**填满四行 cache**，则每 8 行的数据则会填满一个完整的 cache。至此之后就会产生冲突。由于 A 数组和 B 数组相连存放，因此 A 数组的每一位和 B 数组的每一位必定冲突，因此在暴力转置的时候可以说是每次访问 A 和 B 数组都一定会产生冲突。为了减少冲突的次数，我们可以一次性访问连续的八个数字，**这八个数字一定是处于同一个 cache 块中的**，因此可以大大减少 miss 的次数。

```
void transpose_32(int M,int N,int A[N][M],int B[M][N]){
 for(int i = 0; i < 32; i += 8)
        for(int j = 0; j < 32; j += 8)
            for (int k = i; k < i + 8; k++)
            {
                int a0 = A[k][j];
                int a1 = A[k][j+1];
                int a2 = A[k][j+2];
                int a3 = A[k][j+3];
                int a4 = A[k][j+4];
                int a5 = A[k][j+5];
                int a6 = A[k][j+6];
                int a7 = A[k][j+7];
                B[j][k] = a0;
                B[j+1][k] = a1;
                B[j+2][k] = a2;
                B[j+3][k] = a3;
                B[j+4][k] = a4;
                B[j+5][k] = a5;
                B[j+6][k] = a6;
                B[j+7][k] = a7;
            }
}
```

图 2-1 32*32 矩阵转置的实现

2.3.2 64*64 矩阵转置

当数据来到 64*64 之后就不再能用 32*32 的 8*8 分块了，**因为每四行就会占满 Cache 的缓存**，所以利用 8*8 分块依旧会有一半的 miss 出现，无法大幅降低 miss 的值。那么能否考虑 4*4 分块呢，毕竟每四行并不会在 Cache 中冲突。我在本地尝试了一下，虽然效果还可以但是并不能得到满分，原因在于 4*4 分块并不能完全利用 Cache 的全部缓存内容，有一大半的 Cache 缓存没有利用上。因此我们就可以考虑这样做：还是用基本的 8*8 分块，不过这次在 8*8 分块中不一次性全部访问完，而是将 8*8 分块中内部再次进行分块分成 4 个 4*4 的小分块，对这四个小分块分别处理：

（1）将 A 的左上角和右上角一次性复制给 B

（2）用本地变量将 B 的右上角存下来

（3）将 A 的左下角复制给 B 的右上角

（4）将本地变量储存的 B 的右上角本地变量复制给 B 的左下角

（5）把 A 的右下角复制给 B 的右下角

这样做就可以将缓存利用完全并且消除同一行中的冲突。

```
void transpose_64(int M,int N,int A[N][M],int B[M][N]){
 int a0,a1,a2,a3,a4,a5,a6,a7;
  if(N == 64){
   for(int i=0;i<64;i+=8){
    for(int j=0;j<64;j+=8){
     for(int x = i;x<i+4;x++){
      a0 = A[x][j+0];
      a1 = A[x][j+1];
      a2 = A[x][j+2];
      a3 = A[x][j+3];
      a4 = A[x][j+4];
      a5 = A[x][j+5];
      a6 = A[x][j+6];
      a7 = A[x][j+7];
      B[j+0][x] = a0;
      B[j+1][x] = a1;
      B[j+2][x] = a2;
      B[j+3][x] = a3;
      B[j+0][x+4] = a4;
      B[j+1][x+4] = a5;
      B[j+2][x+4] = a6;
      B[j+3][x+4] = a7;
     }
     for(int x=j;x<j+4;x++){
      a0 = B[x][i+4];
      a1 = B[x][i+5];
      a2 = B[x][i+6];
      a3 = B[x][i+7];
      a4 = A[i+4][x];
      a5 = A[i+5][x];
      a6 = A[i+6][x];
      a7 = A[i+7][x];
      B[x][i+4] = a4;
      B[x][i+5] = a5;
      B[x][i+6] = a6;
      B[x][i+7] = a7;
      B[x+4][i+0] = a0;
      B[x+4][i+1] = a1;
      B[x+4][i+2] = a2;
      B[x+4][i+3] = a3;
     }
     for(int k = i+4;k < i+8;k++){
      a1 = A[k][j+4];
      a2 = A[k][j+5];
      a3 = A[k][j+6];
      a4 = A[k][j+7];
      B[j+4][k] = a1;
      B[j+5][k] = a2;
      B[j+6][k] = a3;
      B[j+7][k] = a4;
     }
    }
   }
}
}
```

图 2-2 64*64 矩阵转置的实现

2.3.3 61*67 矩阵转置

61*67 矩阵的实现比较简单，分别尝试一下 4*4, 8*8, 16*16 的分块即可，最后发现 16*16 的矩阵可以通过测试，达到了题目要求。不过需要注意的一点就是列数和行数并不是 16 的倍数，对于这部分采用简单的朴素做法也就是暴力判断是否越界，没有越界则正常使用反之则跳过即可。

```
void transpose_61(int M,int N,int A[N][M],int B[M][N]){
 for (int i = 0; i < N; i += 16)
        for (int j = 0; j < M; j += 16)
             for (int k = i; k < i + 16 && k < N; k++)
               for (int s = j; s < j + 16 && s < M; s++)
                    B[s][k] = A[k][s];
}
```

图 2-3 61*67 矩阵转置

2.4. 实验结果和分析

通过头哥测试，结果无误并且成功获得满分：

测试矩阵转置

实验汇总:

转置矩阵类型	得分	该项分值	是否有效
32x32	8.0	8	287
64x64	8.0	8	1227
61x67	10.0	10	1992
合 计	26.0	26	

矩阵32x32转置优化完成,
矩阵64x64转置优化完成,
矩阵61x67转置优化完成!

图 2-4 通过头哥测试并且获得满分

3. 总结和体会

本次实验主要围绕组相联 Cache 模拟器的设计与矩阵转置算法的 Cache 优化两项任务展开，旨在加深对计算机体系结构中 Cache 存储系统运行机制的理解，并通过实践提升对程序性能优化方法的掌握程度。实验过程不仅涵盖了底层存储结构的建模与仿真，更涉及对高层算法与底层硬件特性之间协同关系的深入分析与优化

在组相联 Cache 模拟器的设计与实现过程中，我系统学习并掌握了 Cache 的基本组成结构和运行原理，包括地址映射方式、数据替换策略、写入策略以及缓存行的组织方式等关键内容。通过编程实现模拟器，我深入理解了如何通过分解内存地址实现组相联映射，如何采用如 LRU（最近最少使用）等策略进行替换操作，以及在读写命中与不命中情况下的不同处理流程。这一过程中，理论知识得到了有效验证

而在矩阵转置优化部分，实验引导我从内存访问局部性（Locality）的角度重新审视算法设计的重要性，我认识到良好的数据访问模式对 Cache 命中率具有决定性影响。通过分析并改进原始算法，显著减少了 Cache 不命中次数，提升了程序运行效率。这使我意识到，优秀的算法设计不仅追求时间复杂度的优化，也应重视数据局部性和内存访问效率。

此外，本次实验加深了我对理论知识与实际应用之间关系的理解。在设计和调试 Cache 模拟器的过程中，我不断遇到理论模型与编程实现之间的差异与挑战，这种实践中的“误差”反过来促使我更加准确地把握理论的边界与应用条件。同时，在优化矩阵转置算法的过程中，我也进一步意识到，程序运行性能的提升并非仅依赖于更快的处理器，而往往源于对内存访问模式、数据布局和缓存机制等系统性因素的综合把控。

4. 对实验课程的建议

（1）希望课程能够给与更多的参考资料和相关内容，仅仅依靠头歌上的部分讲解还是难以理解实验的具体要求。

（2）希望课程能够完善本地运行的环境配置教程。不要仅仅给一个头歌环境即可，很多调试等功能还是在本地更好进行。

（3）希望可以增加 Cache 的替换策略选择性，可以增加 FIFO，近似 LRU 等算法的实验并比较这几种算法的差异性，这样可以增加同学们对 Cache 知识的掌握。

（4）对优化矩阵转置实验的讲解还是过于简单，引导性比较弱，初读的时候有些云里雾里。希望能够稍微精炼一下语言详细描述一下该实验的思想或者是直接给出 32*32 的具体思路让学生自主思考后续 64*64 实验和 61*67 的方法。

作者签名：