华中科技大学

计算机系统结构实验报告

姓　　名：　　董玲晶

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2005

学　　号：　　U202090063

分数	
教师签名	

2023 年. 5 月. 19 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.....3
1.4. 实验结果和分析.....9

2. 优化矩阵转置实验.....10

2.1. 实验目的.....10
2.2. 实验环境.....10
2.3. 实验思路.....10
2.4. 实验结果和分析.....12

3. 总结和体会.....13

4. 对实验课程的建议.....13

1. Cache 模拟器实验

1.1. 实验目的

- 理解 cache 工作原理
- 实现一个高效的模拟器

1.2. 实验环境

Educoder 实验模拟环境。编程语言：C 语言；操作系统：Linux 系统

1.3. 实验思路

1.3.1 设计 Cache 模拟器的数据结构

定义如下数据结构：

- cache_line_t：Cache 行的结构体，包括有效位、标记位和 LRU 计数器；
- cache_set_t：Cache 组的类型定义，一组 Cache 行（cache_line_t）的指针；
- cache_t：Cache 的类型定义，一组 Cache 组（cache_set_t）的指针。

代码 1：Cache 模拟机数据结构

```
typedef struct cache_line {
  char valid;   //有效位
  mem_addr_t tag;    //标记位
  unsigned long long int lru;   //LRU 计数器
}
cache_line_t;
typedef cache_line_t * cache_set_t;
typedef cache_set_t * cache_t;
```

1.3.2 参数

定义全局变量、命令行参数派生的变量以及用于记录缓存统计信息的计数器。全局变量用于存储命令行参数设置和其他全局状态，命令行参数派生的变量用于计算组数和块大小，而缓存统计信息的计数器用于记录缓存的命中、未命中和逐出次数。代码如下：

代码 2：Cache 模拟机数据结构

```
/* 全局变量由命令行参数设置 */
int verbosity = 0; //如果设置，则打印跟踪
int s = 0; //组索引位数
int b = 0; //块偏移位数
int E = 0; //相联度
char * trace_file = NULL;

/* 命令行参数 */
int S; //组数
int B; //块大小（字节）

/* 用于记录缓存统计信息的计数器 */
int miss_count = 0; //缺失计数器
int hit_count = 0;   //命中计数器
int eviction_count = 0; //逐出计数器
unsigned long long int lru_counter = 1;  //LRU 计数器

/* cache */
cache_t cache;
mem_addr_t set_index_mask;
```

其中，verbosity 用于控制是否打印跟踪信息。当设置为非零值时，会打印详细的跟踪信息；s、b、E 分别表示组索引位数、块偏移位数和相联度；trace_file 一个字符指针，用于存储跟踪文件的名称。S 表示组数，根据组索引位数 s 计算得出；B 表示块大小，根据块偏移位数 b 计算得出。第三部分的计数器变量用于跟踪 Cache 的性能和统计信息，具体作用见代码注释。最后设置用于提取组索引的掩码 set_index_mask，用于存储组索引位的掩码变量，即使用一定数量的组索引位来确定数据应存储在缓存中的哪个组

1.3.3 initCache() 函数

该函数负责 Cache 数据结构初始化，用于处理后续的访存操作。根据给定的组数 S 和相联度 E 动态分配内存空间，并将每个 Cache 行的有效位、标记位和 LRU 计数器初始化为零。

1.3.4 freeCache() 函数

该函数负责释放之前已分配的 Cache 内存空间：遍历每个组，释放每个组中 Cache 行的内存空间，最后释放 Cache 的内存空间。

1.3.5 accessData(mem_addr_t addr) 函数

该函数负责对数据的访问操作。它接收一个内存地址 addr，判断是否命中或者是否淘汰，然后执行 Cache 的命中、缺失和逐出等操作。代码如下：

代码 3：accessData ()函数

```
void accessData(mem_addr_t addr) {
  int flag = 0, goal = 0;
  mem_addr_t tag_now = (addr >> b) >> s;
  set_index_mask = (addr >> b) & ((1 << s) - 1);
  for (int i = 0; i < E; ++i) {
    if (cache[set_index_mask][i].valid == 1 &&
        cache[set_index_mask][i].tag == tag_now) {
      hit_count++; lru_counter++;
      cache[set_index_mask][i].lru = lru_counter; return;
    }
  }
  miss_count++;
  for (int i = 0; i < E; ++i) {
    if (cache[set_index_mask][i].valid == 0) {
      flag = 1; goal = i; break;
    }
  }
  if (flag == 1) {
    cache[set_index_mask][goal].valid = 1;
    cache[set_index_mask][goal].tag = tag_now;
    cache[set_index_mask][goal].lru = (++lru_counter);
  } else {
    eviction_count++;
    flag = 0;
    int minlru = cache[set_index_mask][0].lru;
    for (int i = 0; i < E; i++) {
      if (minlru > cache[set_index_mask][i].lru) {
        minlru = cache[set_index_mask][i].lru; flag = i;
      }
    }
    cache[set_index_mask][flag].valid = 1;
    cache[set_index_mask][flag].tag = tag_now;
    cache[set_index_mask][flag].lru = (++lru_counter);
  }}
```

首先根据地址计算出组索引和标签。然后，遍历当前组中的缓存行，检查是否存在与给定标签相匹配的缓存行。如果找到匹配的缓存行，则表示缓存命中，相应的计数器增加并返回。如果没有找到匹配的缓存行，则发生缺失。在缺失情况下，函数继续检查是否有空闲的缓存行可用，如果有，则将数据放入空闲行中。如果没有空闲行可用，则需要选择一个最近最少使用的缓存行进行替换。找到最小的 LRU 计数器值的缓存行后，将数据放入该缓存行，并更新相关计数器。最终，函数返回并完成了对数据的访问操作。

1.3.6 replayTrace(char * trace_fn) 函数

该函数负责读取 trace 轨迹文件的内容，并根据其指令进行模拟内存访问的过程。代码如下：

代码 4：replayTrace()函数

```
void replayTrace(char * trace_fn) {
  char buf[1000]; mem_addr_t addr = 0; unsigned int len = 0;
  FILE * trace_fp = fopen(trace_fn, "r");
  if (!trace_fp) {
    fprintf(stderr, "%s: %s\n", trace_fn, strerror(errno));
    exit(1);
  }
  while (fgets(buf, 1000, trace_fp) != NULL) {
    if (buf[1] == 'S' || buf[1] == 'L' || buf[1] == 'M') {
      sscanf(buf + 3, "%llx,%u", & addr, & len);
      if (verbosity)
        printf("%c %llx,%u ", buf[1], addr, len);
      accessData(addr);
      /* 如果指令为读写，则再次访问 */
      if (buf[1] == 'M')
        accessData(addr);
      if (verbosity)
        printf("\n");
    }
  }
  fclose(trace_fp);
}
```

replayTrace 函数打开指定的跟踪文件，并按行读取文件内容。对于每一行数据，解析出访问类型、地址和长度信息，并调用 accessData 函数模拟缓存访问操作。函数支持读和写操作，对于写操作，会执行两次缓存访问操作。在读取过

程中，如果启用了详细输出模式，会打印相关信息。最后，函数关闭跟踪文件，完成缓存访问的模拟。

1.3.7 printUsage(char * argv[]) 函数

该函数负责输出程序的使用说明和选项信息，帮助用户正确使用程序并设置相应的缓存参数。用户可以通过命令行选项来指定组索引位数、每组的行数、块偏移位数和跟踪文件路径。函数还提供了示例用法以供参考。调用 exit(0) 使程序正常退出。代码如下：

代码 5：printUsage ()函数

```
void printUsage(char * argv[]) {
  printf("Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n", argv[0]);
  printf("Options:\n");
  printf("  -h         Print this help message.\n");
  printf("  -v         Optional verbose flag.\n");
  printf("  -s <num>   Number of set index bits.\n");
  printf("  -E <num>   Number of lines per set.\n");
  printf("  -b <num>   Number of block offset bits.\n");
  printf("  -t <file>  Trace file.\n");
  printf("\nExamples:\n");
  printf("  linux>  %s -s 4 -E 1 -b 4 -t traces/yi.trace\n", argv[0]);
  printf("  linux>  %s -v -s 8 -E 2 -b 4 -t traces/yi.trace\n", argv[0]);
  exit(0);
}
```

选项如下：

-h：打印帮助信息。

-v：可选的详细输出标志。

-s <num>：组索引位数。

-E <num>：每组的行数。

-b <num>：块偏移位数。

-t <file>：跟踪文件路径。

1.3.8 main(int argc, char * argv[]) 函数

负责解析命令行参数、初始化缓存、执行缓存访问操作、释放内存和打印统

计信息。通过命令行参数设置缓存的组索引位数 s、块偏移位数 b、相联度 E 和跟踪文件路径 trace_file。然后根据这些参数计算组数 S 和块大小 B。接下来，通过调用 initCache 函数初始化缓存，并使用 replayTrace 函数执行缓存访问操作。执行完毕后，释放分配的内存，并使用 printSummary 函数打印统计信息。具体代码如下，详情见注释。

代码 6：main ()函数

```
int main(int argc, char * argv[]) {
  char c;
  while ((c = getopt(argc, argv, "s:E:b:t:vh")) != -1) {
    switch (c) {
      case 's':
        s = atoi(optarg); break;
      case 'E':
        E = atoi(optarg); break;
      case 'b':
        b = atoi(optarg); break;
      case 't':
        trace_file = optarg; break;
      case 'v':
        verbosity = 1; break;
      case 'h':
        printUsage(argv); exit(0);
      default:
        printUsage(argv); exit(1);
    }
  }
  /* 所有必需的命令行参数都已指定 */
  if (s == 0 || E == 0 || b == 0 || trace_file == NULL) {
    printf("%s: Missing required command line argument\n", argv[0]);
    printUsage(argv);
    exit(1);
  }
  /* 从命令行参数计算 S、E 和 B */
  S = 1 << s;
  B = 1 << b;
  E = E;
  /* 初始化缓存 */
  initCache();
  replayTrace(trace_file);
  /* 释放分配的内存 */
  freeCache();
```

```
    /* 输出自动测试程序的命中和未命中统计信息 */
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}
```

1.4. 实验结果和分析

修改完成后点击评测，得到结果如下，成功通关。

- 对于 yi2.trace、yi1.trace 和 dave.trace 三个跟踪文件，可以看到测试通过，即程序的运行结果与参考模拟器的结果完全一致。
- 对于 trans.trace 跟踪文件，不同的缓存配置产生了不同的命中数、未命中数和逐出数，但是程序的运行结果与参考模拟器的结果完全一致。这说明程序能够正确模拟缓存的行为，对不同的访问模式和缓存配置做出了正确的响应。
- 对于 long.trace 跟踪文件，可以看到命中数、未命中数和逐出数都较大，说明该跟踪文件包含了大量的内存访问操作。程序的运行结果与参考模拟器的结果完全一致，这证明程序能够处理大规模的跟踪文件并正确模拟缓存的行为。

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

图 1：Cache 模拟实验评测结果

2. 优化矩阵转置实验

2.1. 实验目的

- 实现一个矩阵转置函数；
- 要求通过优化缓存访问模式，尽量减少缓存未命中的次数，以提高转置函数的性能。

2.2. 实验环境

Educoder 实验模拟环境。编程语言：C 语言；操作系统：Linux 系统

2.3. 实验思路

对于 64x64 大小的矩阵，采用一种分块的转置策略，利用缓存和硬件并行性来提高性能；对于其他大小的矩阵，采用较简单的转置策略，直接遍历矩阵元素进行转置。

2.3.1 64×64 矩阵

使用两个嵌套的循环来遍历矩阵的块。外层循环通过 i 变量从 0 到 M/8 遍历，对矩阵的行进行分块处理；内层循环通过 j 变量从 0 到 N/8 遍历，对矩阵的列进行分块处理。后续使用两个额外的循环用于处理矩阵的剩余部分：第一个额外循环将矩阵 A 中剩余的列复制到矩阵 B 中对应的行中，第二个额外循环将矩阵 A 中剩余的行复制到矩阵 B 中对应的列中。代码框架如下。

代码 7：整体分块策略框架函数

```
for(i = 0; i < M / 8; i++)
  for(j = 0; j < N / 8; j++)
    ……

for(i = 0; i < N ; i++)
  for(j = 8 * (M / 8); j < M; j++) {
    tmp0 = A[i][j];
    B[j][i] = tmp0;
  }
for(i = 8 * (N / 8); i < N; i++)
```

```
for(j = 0; j < 8*(M/8); j++) {
  tmp0 = A[i][j];
  B[j][i] = tmp0;
}
```

在上述所说的双层嵌套循环内部，采用四个循环对矩阵进行转置。

第一个循环将 A 矩阵的前四列元素（A[j*8+k][i*8+0]到 A[j*8+k][i*8+3]）分别复制到 B 矩阵的不同位置，将 tmp0 赋值给 B[i*8+0][j*8+k]，将 tmp1 赋值给 B[i*8+1][j*8+k]……最终实现将 A 矩阵的前四列转置到 B 矩阵的对应位置。

第二个循环将 A 矩阵的第五到第八列元素（A[j*8+k][i*8+4+0] 到 A[j*8+k][i*8+4+3]）分别复制到 B 矩阵的不同位置，类似于第一个循环，将 A 矩阵的后四列转置到 B 矩阵的对应位置。

第三个循环是一个交换操作，实现了矩阵 B 的后四列与 A 矩阵的后四列的交换。首先，将 B 矩阵中第四列之后的元素赋值给临时变量 tmp0 到 tmp3。然后，将 A 矩阵中第五列之后的元素赋值给临时变量 tmp4 到 tmp7。接下来，通过交换操作，将 tmp0 到 tmp3 的值赋给 B 矩阵中第四列之后的位置，将 tmp4 到 tmp7 的值赋给 B 矩阵中第五列之后的位置。

第四个循环将 A 矩阵的后四列的元素赋值给 B 矩阵中对应位置。类似于第一个循环，将 A 矩阵的后四列转置到 B 矩阵的对应位置。

详细代码如下。

代码 8：具体分块处理转置细节代码

```
for(k = 0; k < 4; k++) {
  tmp0 = A[j*8+k][i*8+0]; tmp1 = A[j*8+k][i*8+1];
  tmp2 = A[j*8+k][i*8+2]; tmp3 = A[j*8+k][i*8+3];
  B[i*8+0][j*8+k] = tmp0; B[i*8+1][j*8+k] = tmp1;
  B[i*8+2][j*8+k] = tmp2; B[i*8+3][j*8+k] = tmp3;
}
for(k = 0; k < 4; k++) {
  tmp0 = A[j*8+k][i*8+4+0]; tmp1 = A[j*8+k][i*8+4+1];
  tmp2 = A[j*8+k][i*8+4+2]; tmp3 = A[j*8+k][i*8+4+3];
  B[i*8+0][j*8+4+k] = tmp0; B[i*8+1][j*8+4+k] = tmp1;
  B[i*8+2][j*8+4+k] = tmp2; B[i*8+3][j*8+4+k] = tmp3;
}
for(k = 0; k < 4; k++) {
  tmp0 = B[i*8+k][j*8+4+0]; tmp1 = B[i*8+k][j*8+4+1];
  tmp2 = B[i*8+k][j*8+4+2]; tmp3 = B[i*8+k][j*8+4+3];
```

```
  tmp4 = A[j*8+4+0][i*8+k]; tmp5 = A[j*8+4+1][i*8+k];
  tmp6 = A[j*8+4+2][i*8+k]; tmp7 = A[j*8+4+3][i*8+k];
  B[i*8+k][j*8+4+0] = tmp4; B[i*8+k][j*8+4+1] = tmp5;
  B[i*8+k][j*8+4+2] = tmp6; B[i*8+k][j*8+4+3] = tmp7;
  B[i*8+4+k][j*8+0] = tmp0; B[i*8+4+k][j*8+1] = tmp1;
  B[i*8+4+k][j*8+2] = tmp2; B[i*8+4+k][j*8+3] = tmp3;
}
for(k = 0; k < 4; k++) {
  tmp0 = A[j*8+4+k][i*8+4+0]; tmp1 = A[j*8+4+k][i*8+4+1];
  tmp2 = A[j*8+4+k][i*8+4+2]; tmp3 = A[j*8+4+k][i*8+4+3];
  B[i*8+4+0][j*8+4+k] = tmp0; B[i*8+4+1][j*8+4+k] = tmp1;
  B[i*8+4+2][j*8+4+k] = tmp2; B[i*8+4+3][j*8+4+k] = tmp3;
}
```

2.3.2 其它大小的矩阵

对于非 64×64 大小的矩阵，只采用较简单的转置策略，直接遍历矩阵元素进行转置。先处理矩阵的整数倍大小的部分，通过三个嵌套循环遍历 A 矩阵中的 8x8 子矩阵，并将子矩阵中的元素逐个赋值给 B 矩阵的对应位置。再通过两个嵌套循环遍历剩余的列，并将 A 矩阵中的元素逐个赋值给 B 矩阵的对应位置，以实现列数不是 8 的倍数的剩余的列的转置。

2.4. 实验结果和分析

修改完成后点击评测，得到结果如下，成功通关。

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	1291
61x67	10.0	10	1875
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：优化矩阵转置实验评测结果

3. 总结和体会

通过 Cache 模拟实验，我深入理解了 Cache 的工作原理和对程序性能的影响，深刻认识到 Cache 对程序性能的重要影响，Cache 的命中与未命中直接影响着程序的执行时间，高命中率可以显著减少内存访问的延迟，提高程序的执行效率。同时，该实验也加深了我对 Cache 工作机制的理解，进一步掌握了 LRU 淘汰算法策略。

而在矩阵转置实验里，我采用了分块的方式进行矩阵转置，并利用临时变量进行数据交换。用这种将矩阵分割成小块的方式，可以充分利用缓存的局部性原理，提高 Cache 命中率，显著降低访存开销，提高程序的执行效率。通过这个实验，我体会到了数据局部性的重要性，这对我今后在编写高效算法和优化程序性能方面将有很大帮助。同时，我也认识到了程序性能优化是一个综合考虑多个方面因素的过程，需要细致地分析问题和不断实践。

4. 对实验课程的建议

（1）或许可以单独安排实验课，安排助教对实验进行答疑和解析，同时进行一些实验指导；

（2）实验确实很经典，但内容有点少了，整个课程其实可以稍微加大一些实验的占比，减少一些理论课课时。