

**计算机系统结构实验报告**

|  |  |
| --- | --- |
| 姓 名： | 瞿明睿 |
| 学 院： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 班 级： | CS2207 |
| 学 号： | U202215561 |
|  |  |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2025 年 5 月15日

**目 录**

[1. Cache模拟器实验 2](#_Toc198321712)

[**1.1.** **实验目的** 2](#_Toc198321713)

[**1.2.** **实验环境** 2](#_Toc198321714)

[**1.3.** **实验思路** 2](#_Toc198321715)

[**1.4.** **实验结果和分析** 3](#_Toc198321716)

[2. 矩阵转置优化 4](#_Toc198321717)

[**2.1.** **实验目的** 4](#_Toc198321718)

[**2.2.** **实验环境** 4](#_Toc198321719)

[**2.3.** **实验思路** 4](#_Toc198321720)

[**2.4.** **实验结果和分析** 5](#_Toc198321721)

[3. 总结和体会 6](#_Toc198321722)

[4. 对实验课程的建议 6](#_Toc198321723)

[电子签名 7](#_Toc198321724)

# Cache模拟器实验

* 1. **实验目的**

编写实现一个Cache模拟器，掌握如何通过程序模拟Cache的行为（命中、缺失以及替换策略）。使用动态内存分配（malloc和calloc）构建通用数据结构以适应不同的 Cache参数配置。根据输入的内存访问轨迹文件，统计并输出Cache的命中次数、缺失次数以及驱逐次数。

* 1. **实验环境**

头歌测试环境，输入数据为内存访问轨迹文件（.trace）

* 1. **实验思路**

本实验模拟的是一个组相联缓存，我们采用采用如下方式组织数据结构

typedef struct {

    int valid;              // 有效

    unsigned long long tag; // 标记位

    unsigned long long last\_used; // 最近使用时间戳（用于 LRU）

} cache\_line;

valid表示该缓存行是否被占用；tag用于判断访问地址是否命中；last\_used记录该缓存行最后一次被访问的时间，用于LRU替换策略。

我们使用一个二维数组结构来表示Cache：第一维表示组，大小为2^8；第二维表示每组中的行，数量为E；在初始化时，根据命令行传入的参数-s,-E,-b动态分配内存：

int S = 1 << s; // 组数 = 2^s

cache = malloc(S \* sizeof(cache\_line\*));

for (int i = 0; i < S; i++) {

    cache[i] = calloc(E, sizeof(cache\_line)); // 每组 E 行

}

同时设计我们的地址，给定32位地址address，将其拆分为三部分：偏移offset共b位，组索引set index共s位，tag为剩余高位。

在程序中通过以下方式提取（掩码方式）

unsigned long long set\_index = (address >> b) & ((1 << s) - 1);

unsigned long long tag = address >> (s + b);

那么我们的整个缓存访问流程如下

查找命中，遍历当前组的所有行，查找valid==1且tag==当前标记位的行，则为命中，更新时间戳。

未命中处理，若存在空闲行（即valid == 0的行），则直接加载该块到Cache中，若没有空闲行，则使用LRU算法选择一个行进行替换，并增加驱逐计数。

特殊操作处理，对于命令行中特殊的字符处理：对于指令类型M（修改），模拟两次访问（先写后读）；忽略所有以 I开头的指令（仅关注数据 Cache），同时需要使用current\_timestamp变量维护全局时间，用于 LRU 判定

使用malloc和calloc动态创建缓存结构，所有动态分配的内存在程序结束时释放，确保程序不会出现内存泄漏。

* 1. **实验结果和分析**

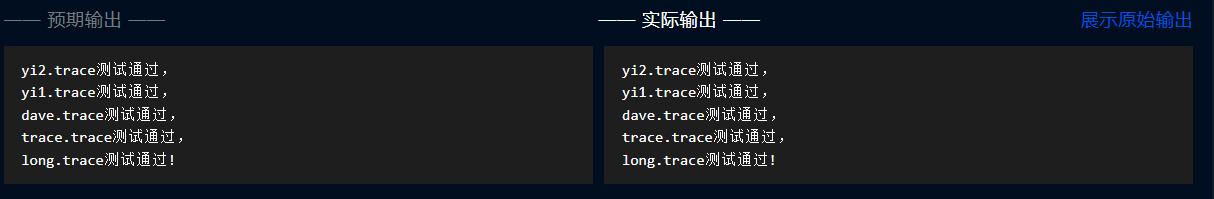


图1 输出

输出正常，如上图

# 矩阵转置优化

* 1. **实验目的**

在trans.c中使用C语言编写一个实现矩阵转置的函数transpose\_submit。使函数调用过程中对cache的不命中数miss尽可能少。

掌握如何通过程序优化减少Cache miss次数；理解数据访问局部性对性能的影响；使用分块技术提高缓存命中率；在有限制条件下（最多只能定义12个int局部变量），实现高效矩阵转置。

* 1. **实验环境**

头歌测试环境，输入数据为三种尺寸的矩阵（32×32、64×64、61×67）

* 1. **实验思路**

原始转置函数采用简单的行优先遍历方式：

for (i = 0; i < N; i++) {

    for (j = 0; j < M; j++) {

        B[j][i] = A[i][j];

    }

}

该方法在访问矩阵A时按行顺序进行，而访问矩阵B时按列顺序进行，导致 Cache miss率高，特别是当矩阵较大时，无法充分利用Cache的空间局部性。

为降低Cache缺失数，采用了以下优化手段：

分块策略：将矩阵划分为多个小块，每次只处理一个块的数据，使得该块的所有数据尽可能驻留在Cache中。在32×32和64×64的情况下使用8×8分块；在61×67 的非规则矩阵中使用16×16分块。

为了减少Cache缺失，将大矩阵划分为若干小块，使得每次只处理一个小块的数据，从而提高局部性。

对于32×32矩阵：使用8×8分块策略，按行读取源矩阵A中的一块数据，一次性写入目标矩阵 B 的列中，这样利用了空间局部性，减少了对A和B的重复访问导致的Cache Miss。

对于64×64矩阵：由于64×64的大小容易造成Cache行冲突未命中，因此采用两阶段访问策略：第一阶段读取前4行，写入B的前4列；第二阶段：读取后4行，同时利用临时变量交换数据，避免重新加载同一行。这种策略显著降低了冲突未命中，提高了性能。

局部变量限制下的优化：受限于最多只能使用12个int类型的局部变量，代码中复用t0~t7 来暂存连续的8个元素，完成一次性的数据搬移，避免多次访问相同内存地址。这种方式既满足了局部变量限制，又提升了命中率。

边界情况处理：对于不能被块大小整除的矩阵维度，添加额外循环处理余下部分，保证所有数据都被正确转置。

* 1. **实验结果和分析**

头哥平台正常过测，内存占用少。

图2 过测

# 总结和体会

本次实验让我深入理解了cache对程序性能的巨大影响。通过优化矩阵转置函数，我掌握了如何利用局部性原理和分块技术来减少 Cache 缺失。受限的局部变量数量要求编写代码时更加注重变量的复用和逻辑设计。

此外，虽然分块算法在理论上能提升性能，但在实际实现中还需要考虑矩阵大小、Cache 配置等因素，比如实际上头歌对于分块是有限制的，当分块过大等原因会导致out of list报错，需要注意。

# 对实验课程的建议

如果能让调试的过程可视化会更好，尤其是对于实验二的调整，经过很多次大的改动我都不知道为什么会list index out iof range,在多次测试我才了解到原因，我同时也希望课程组能够提供完整的线下包来让学生实验，这样可做调整的地方更多找问题也会更快些.

# 电子签名

