

**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](#_Toc13239)

[1.1. 实验目的 3](#_Toc10680)

[1.2. 实验环境 3](#_Toc29165)

[1.3. 实验思路 4](#_Toc2645)

[1.3.2 Cache的初始化及销毁 5](#_Toc24311)

[1.3.2 Cache的访问，更新和替换 5](#_Toc29875)

[1.3.2 Cache的命令行解析 6](#_Toc28933)

[1.4. 实验结果和分析 7](#_Toc6701)

[2. 优化矩阵转置实验 8](#_Toc75)

[2.1. 实验目的 8](#_Toc13666)

[2.2. 实验环境 8](#_Toc31547)

[2.3. 实验思路 8](#_Toc26625)

[2.3.1 32\*32矩阵转置 8](#_Toc23980)

[2.3.2 64\*64矩阵转置 9](#_Toc9187)

[2.3.3 61\*67矩阵转置 10](#_Toc2564)

[2.4. 实验结果和分析 11](#_Toc25088)

[3. 总结和体会 12](#_Toc23773)

[4. 对实验课程的建议 13](#_Toc15877)

# Cache模拟器实验

## 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

## 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 |

## 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**

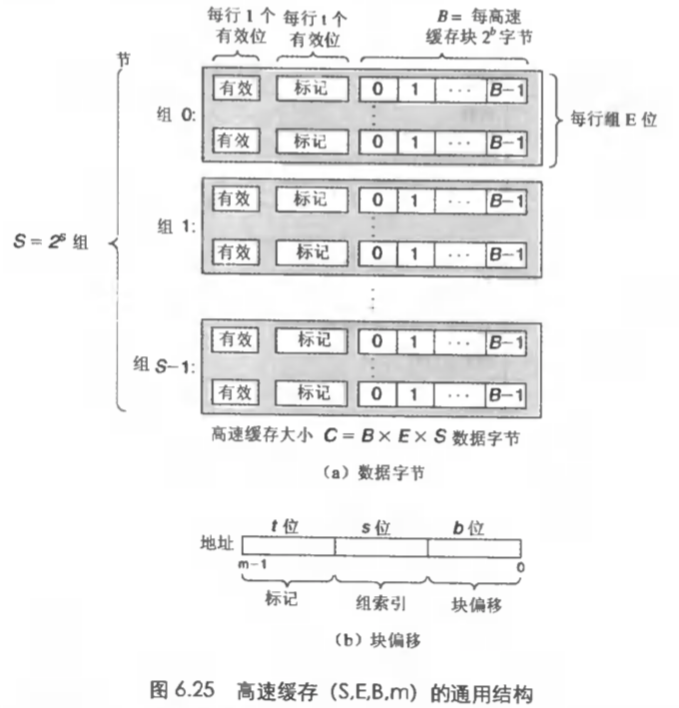


图 1-1 Cache结构图示

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。

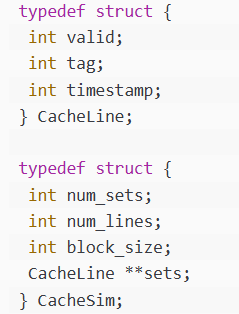


图 1-2 Cache结构设计

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。

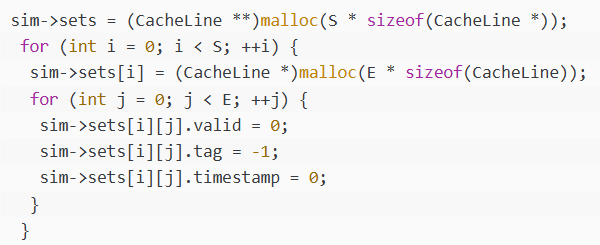


图 1-3 Cache的初始化malloc

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。

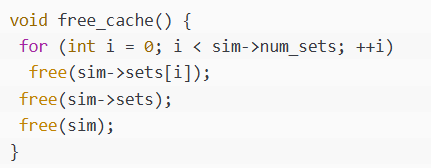


图 1-4 Cache的销毁

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。

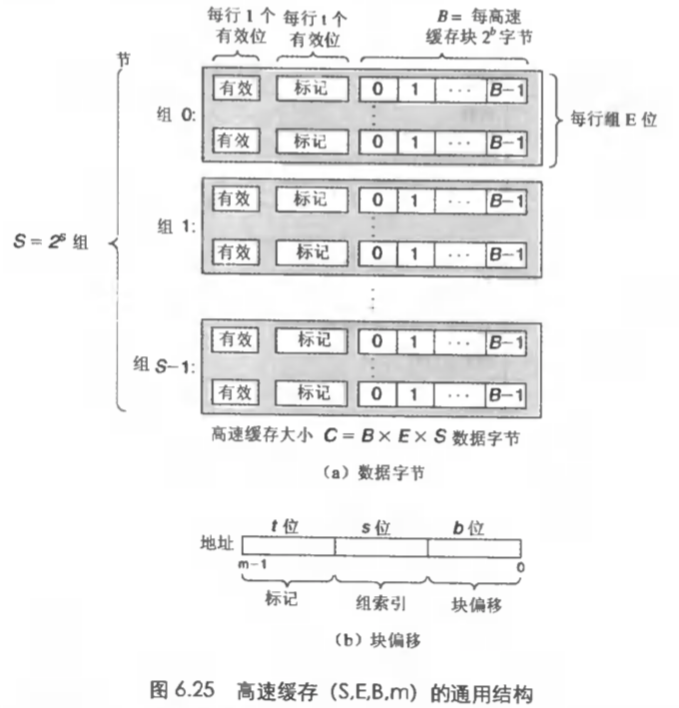


图 1-5 地址对应的信息

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。

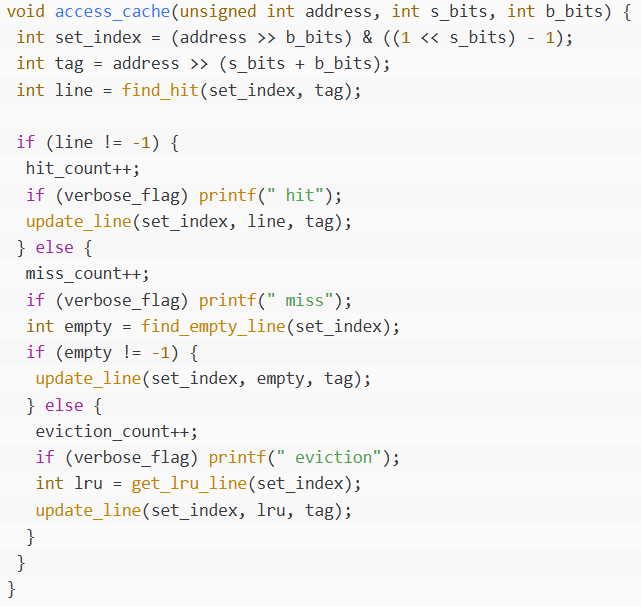


图 1-6 根据对应地址求出相应信息访问Cache中对应行

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。

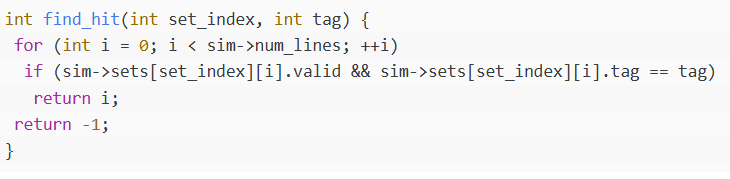


图 1-7 find\_hit函数作为例子，上述四个函数方法均类似该函数

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。

图 1-8 命令行解析函数

## 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：

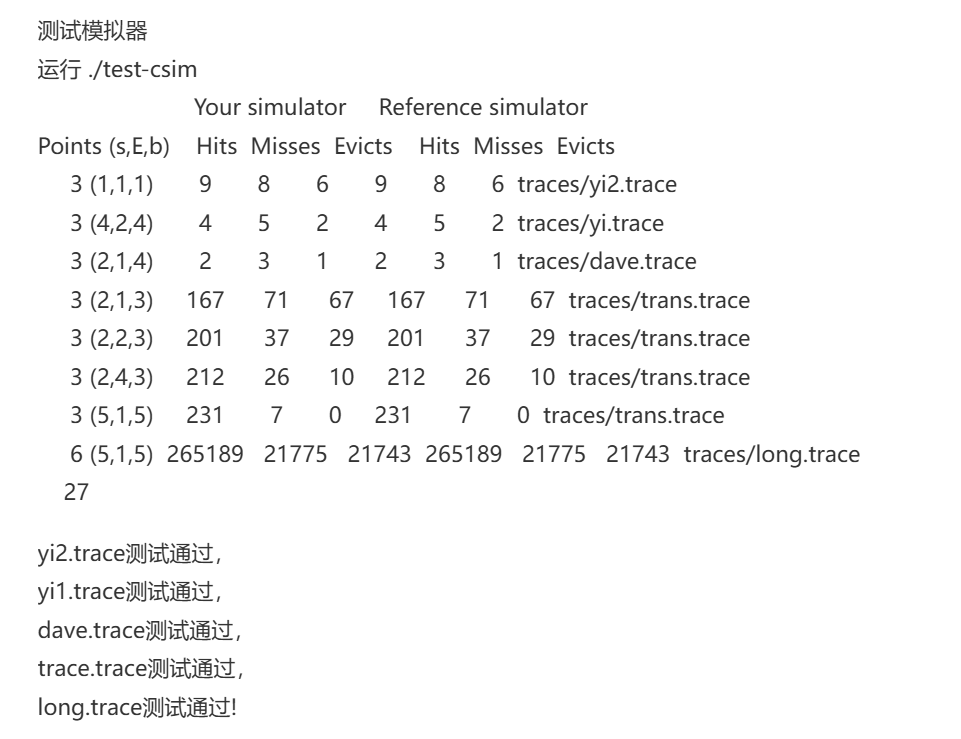


图 1-9 头歌测试通过

# 优化矩阵转置实验

## 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。

* 1. **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 |

* 1. **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。

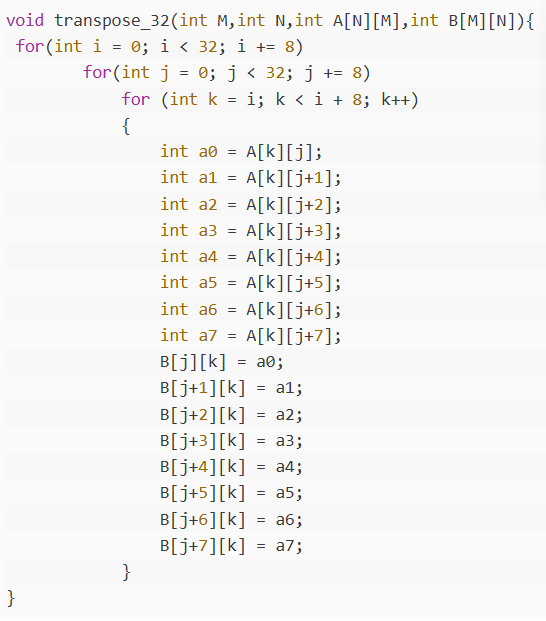


图 2-1 32\*32矩阵转置的实现

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。

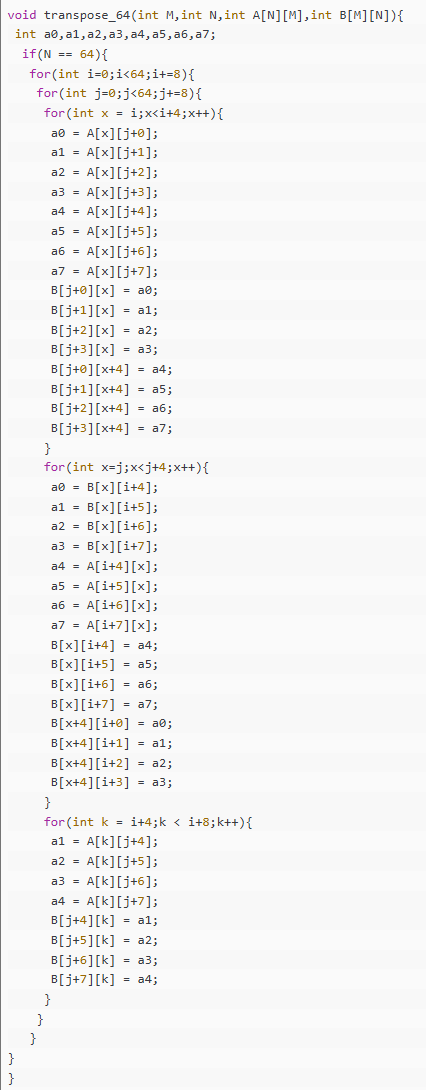


图 2-2 64\*64矩阵转置的实现

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。

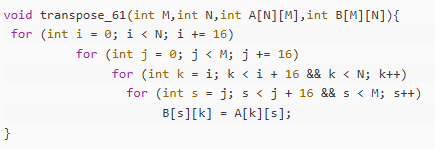


图 2-3 61\*67矩阵转置

* 1. **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：

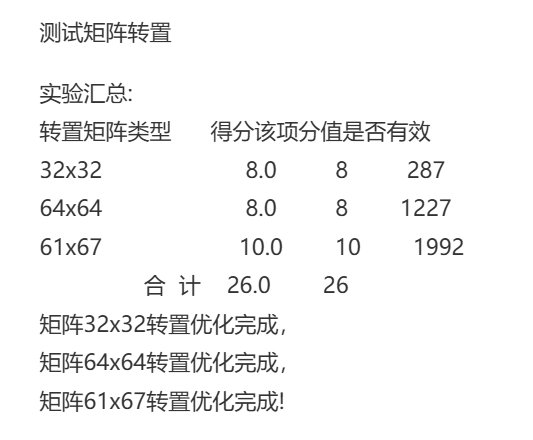


图 2-4 通过头哥测试并且获得满分

# 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。

# 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。

1. 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。
2. 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。
3. 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：asd**