**2021-2021-(2)-计算机系统基础**

《计算机系统基础》

实验报告\_5

成 绩：

评阅人：

学院：智能与计算学部

专业： 新工科试验班

年级：2020级

班级：1班

姓名：吉祥宇

学号：3020244131

日期：2021年5月12日

1. 实验题目及目标

***缓存实验***

*掌握高速缓存的基本工作方式，理解高速缓存对于程序性能的影响。*

1. 实验内容/问题描述：

***本实验分为两部分，分别要求模拟一个高速缓存和利用高速缓存实现更高效的程序。***

***任务A高速缓存模拟：****本部分要求实现一个高速缓存模拟器，可以根据给定参数实现任意类型高速缓存的存储模拟，并给出数据加载信息。*

***任务B矩阵转置：****本部分要求利用给出的高速缓存模拟器实现一个矩阵转置的程序，尽可能减少缓存不命中次数。*

1. 问题分析：

*对于A部分，采取模拟的办法，根据内存加载指令，动态分配和调整空间标识，以模拟块加载和驱逐。在驱逐规则上，采取LRU方式。*

*对于B部分，先实现一个可行的转置程序，再分析其中引起缓存不命中的最大原因，并逐步优化程序。*

**任务A环境：**

***实验平台：虚拟仿真实验平台***

***平台环境：***

|  |  |
| --- | --- |
| **操作系统** | Linux Ubuntu 18.02 |
| **ISA** | x86-64 |
| **CPU核心数** | 2 |
| **内存** | 2G |
| **硬盘** | 50G |

**任务B环境：**

***实验平台：VMware WorkStation Pro虚拟机平台***

***平台环境：***

|  |  |
| --- | --- |
| **操作系统** | Linux Ubuntu 20.04 |
| **ISA** | x86-64 |
| **CPU核心数** | 2 |
| **内存** | 4G |
| **硬盘** | 30G |

***其他约定：***

*对于A部分：*

1.仅考虑数据加载，指令加载指令可以忽略。

2.数据满足对齐规则，长度字段可以忽略。

*对于B部分：*

1.不可更改A数组内容，但可随意更改B数组内容。

2.仅允许使用12个int类型局部变量。

1. 实验步骤/解题思路详解：

**（1）文件结构和概况**

本实验文件目录内文件较多，仅关注重点内容。

文件目录如下：

**-lab5**

**--csim.c**

**--trans.c**

**--csim-ref**

**--test\_trans.c**

**--driver.py**

**…**

其中前两个文件为代码文件，其余为测试工具和辅助文件。

**（2）任务A**

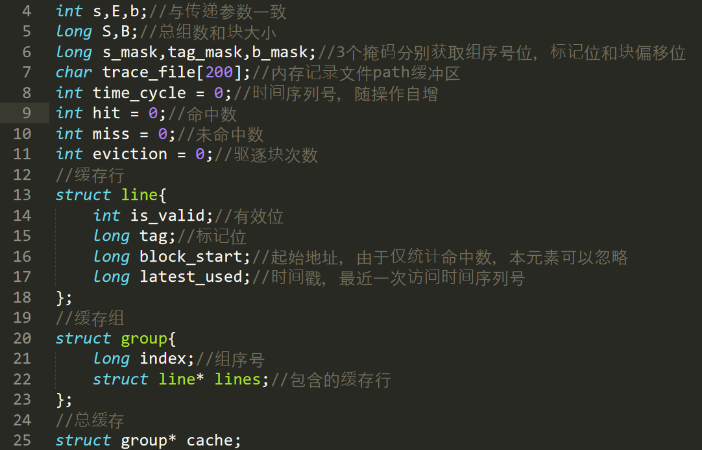
对于模拟高速缓存，可以采取模拟的算法，根据参数要求，动态分配空间。同时，利用行内时间戳记录该缓存行最近一次被访问到的时间，以实现LRU算法。

**说明：为方便编写实验报告，在运行成功的代码文件基础上，编写实验报告时，在代码内部增加了若干注释，不影响程序的正确性。**

**a.存储器结构模拟**

作为一个模拟器，显然需要设计合适的数据结构模拟高速缓存。

根据真实高速缓存，设计如下的数据结构：



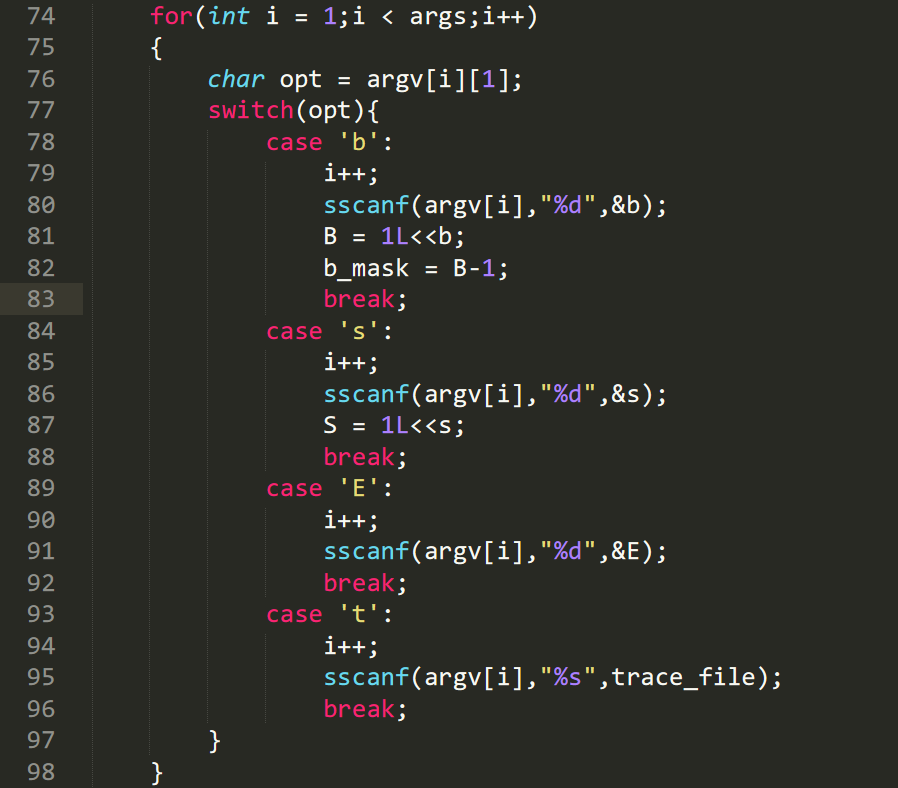
**（图1：数据结构设定和全局变量说明）**

**b.参数处理和初始化**

此部分的主要目的是读取调用程序时传递的参数，以便生成对应结构的模拟缓存。

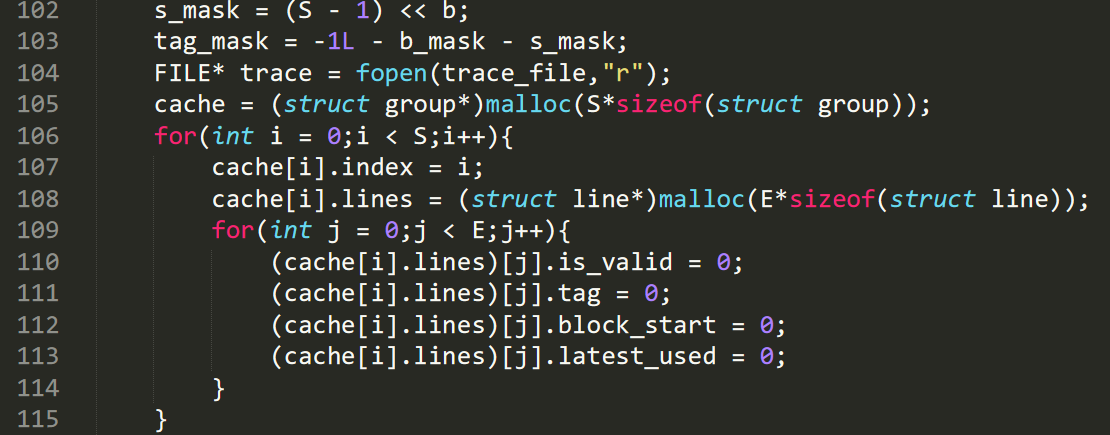
对于参数处理，注意第一个参数为程序路径不做处理，其余的参数按照两个一组分组，获取参数选项，并根据具体选项将值读取至对应变量内。

利用实验3中查询到的库函数sscanf()，可以方便的将字符串内的数字读入。



**（图2：参数处理并初始化缓存参数）**

同时根据读入结果，动态分配内存空间模拟缓存结构，并设定获取组序号、标记位、块偏移对应位的掩码。打开文件输入流，准备读取跟踪信息。



**（图3：结构初始化，掩码初始化，文件输入流初始化）**

**c.跟踪信息读取和处理**

在初始化完成后，需要读入内存跟踪信息，并进行模拟。对于任何指令，都需要进行数据的加载，并判断是否命中。因此，我们将其封装为一个功能函数load()，以备调用。

对于不同指令，我们需要进行如下操作：

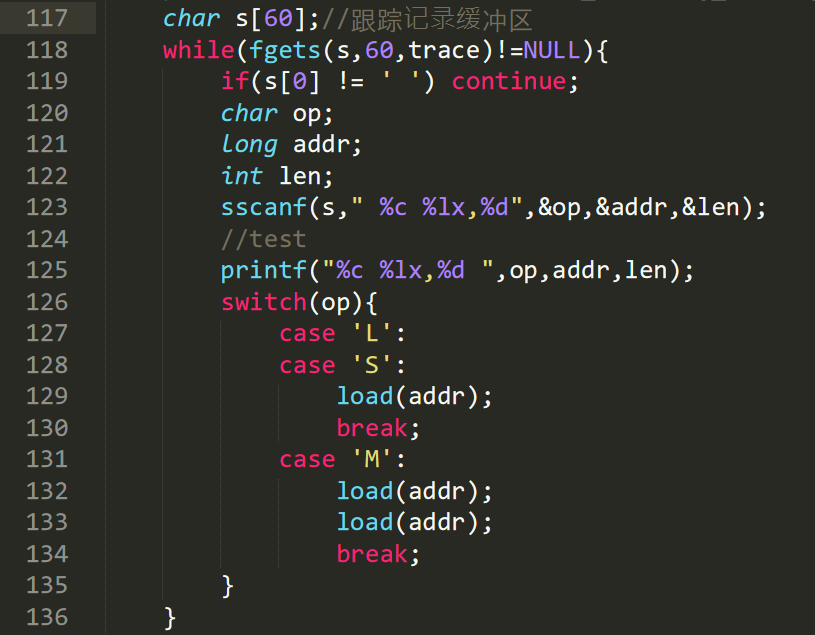
1）I指令：忽略，并直接读取下一行。

2）L指令：加载一次load()。

3）S指令：数据写回，在缓存层面上与L指令一致，调用一次load()。

4）M指令：加载和写回，调用2次load()。

根据如上分析，进行如下的判断：



**（图4：内存跟踪记录中不同类型指令的判断和对应操作）**

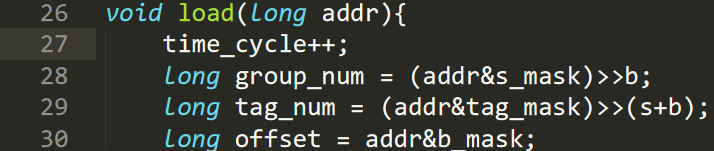
**d. load()函数实现**

在做完了上述准备工作后，我们开始实现主功能函数load()。

显然load仅需一个参数，即地址。

在开始一个新操作前，时间序列需要自增，以表示访问先后顺序。

对于地址，首先要将其分解为3段序列：标记位，组序号和块偏移。



**（图5：地址信息的拆分）**

在获取了这些信息之后，即可以按照真实缓存的访问顺序进行访存了：

**获取组序号对应的组。**

**查找组内是否存在有效位为1且标记相等的缓存行：**

**1)若存在，缓存命中。**

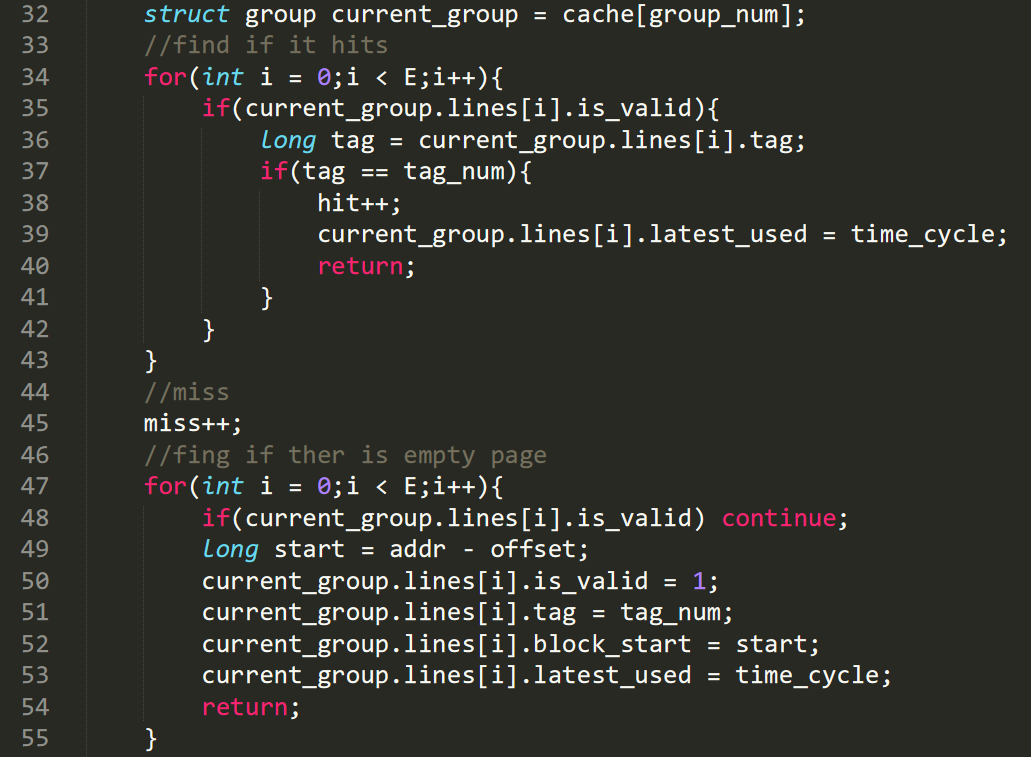
**2)若不存在：**

**a)若组内存在空行，则存入空行**

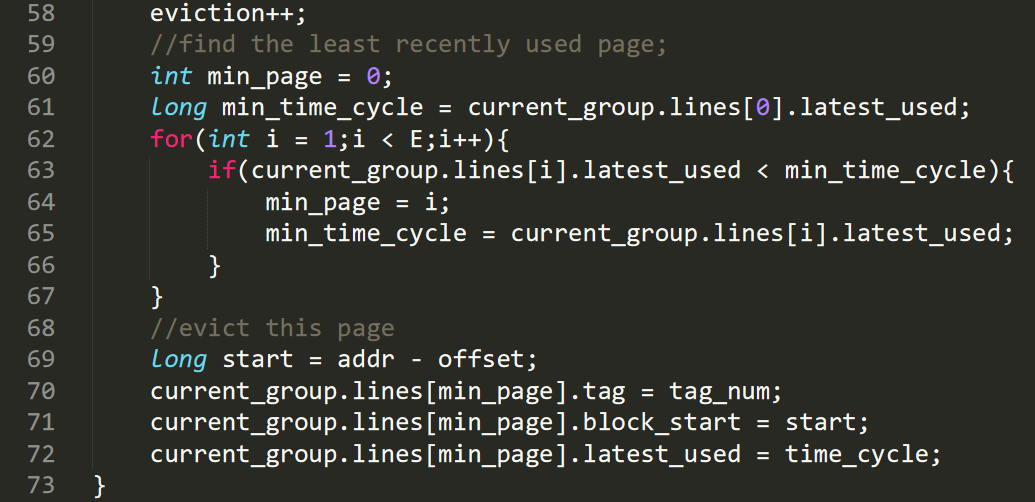
**b)若组内不存在空行，则选择访问时间距离现在最远的一行（即行内时间序列号最小的）驱逐。**

**更新目标行的时间序列号。**

根据上述思路实现对应的代码。



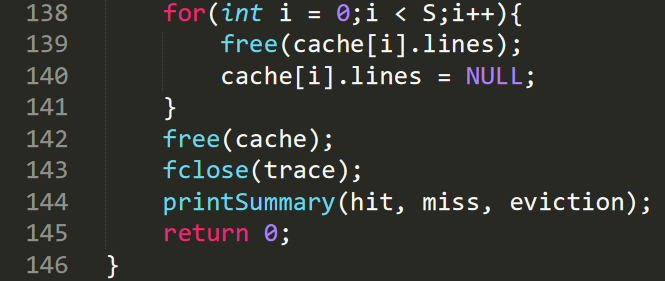
**（图6-1：寻找缓存行1-缓存命中或缓存不命中但存在空行时）**



**（图6-2：寻找缓存行2-缓存不命中且不存在空行）**

**e.资源回收**

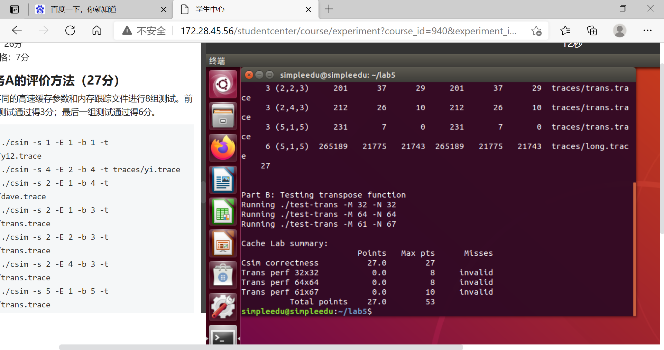
由于模拟器内存时动态分配的，在程序运行结束时，需要手动回收内存。



**（图7：动态内存回收）**

**f.运行结果**

运行程序验证，结果正确。



**（图8：任务A运行通过）**

**（3）任务B**

对于本任务，我们先对样例进行分析，找到影响不命中次数最大的因素，并根据自身特点，进行优化。对于任意版本的转置函数，也采用上述分析方法。

**a.任务约定**

由于需要本任务需要针对不同类型的矩阵进行高速缓存优化，而不同矩阵有着不同的结构特点，实现统一规则几乎不可行。参照CSAPP的英文实验手册，本任务仅需要对给定的3个规模进行优化，并满足要求的不命中数限制即可。

给定的3个规模如下：

***M = 32，N = 32，miss<300***

***M = 64，N = 64，miss<1200***

***M = 61，N = 67，miss<2000***

**b.样例分析**

给定样例是最常见的原始转置方式，即将A的一行同一复制到B的一列中。显然，由于行序优先的存储方式，会导致B的每一次写入均为不命中。

但是，对于矩阵转置，又不得不使用列优先访问模式。在列优先访问模式下，避免不命中的方式只能是将全部数据存入高速缓存中。因此，我们需要进行分块进行转置。

**c.缓存结构分析**

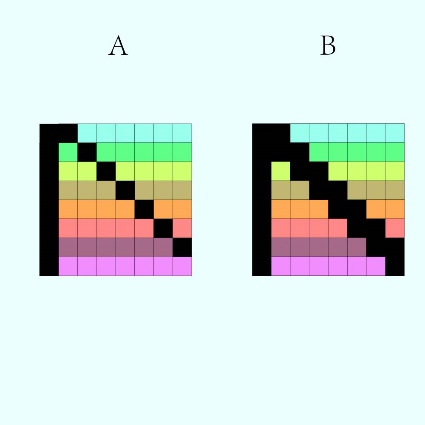
根据手册，本任务缓存器结构为s = 5,E = 1,b = 5的直接映射高速缓存。对于每一个缓存行，可以存储8个数组元素。对于整个高速缓存，可以存储32个缓存行。

**d.32\*32分析**

对于32\*32的矩阵，每一行占用4个缓存行，整个高速缓存可以存储8行，而每一个缓存行正好存储8个元素。因此，我们对A采用8\*8分块进行转置。每一分块内采用原始转置方案进行。

经过验证，不命中次数下降到了343次，未达到要求，仍需继续分析分块时的缓存命中情况。由于缓存结构，A数组的元素和B数组相同位置的元素被映射到了相同缓存块内。而在对角线的8\*8块转置时，A块和B块对应的缓存一致，会引发冲突不命中。而这则是分块中引发不命中的主要原因，在对角线处这一现象极为明显。因此，在后续的分析中，如未加说明，则默认分析第一个分块（即A左上角分块）。

将分块进行分析可以得到如下结果：

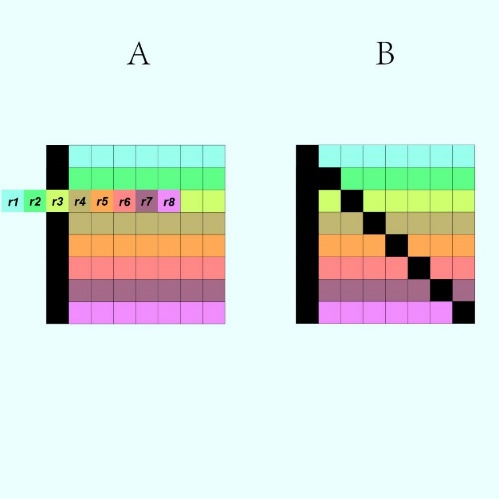


**（图9:32\*32矩阵转置原始8\*8分块缓存分析）**

**说明：缓存分析图中，每一个格子代表一个数组元素，同一颜色格子代表映射到同一缓存行，彩色表示命中，黑色表示未命中。未加说明时，以下分析图均采用上述约定。**

可以发现，由于冲突不命中，当第一次将B的所有元素加载至高速缓存中时，访问下一行的A数组元素时，必须将B的某一行驱逐。然而在同一时刻，又需要将B数组元素加载回缓存中以便存放转置结果，于是引发了大量的不命中。

考虑到我们仍然剩余一些可以使用的局部变量，可以利用剩余的局部变量先行缓存A分块的一行，。此时再去赋值，B的对应列就可以从局部变量中取值，而无需从A中读取，避免了上述互相驱逐的情况。

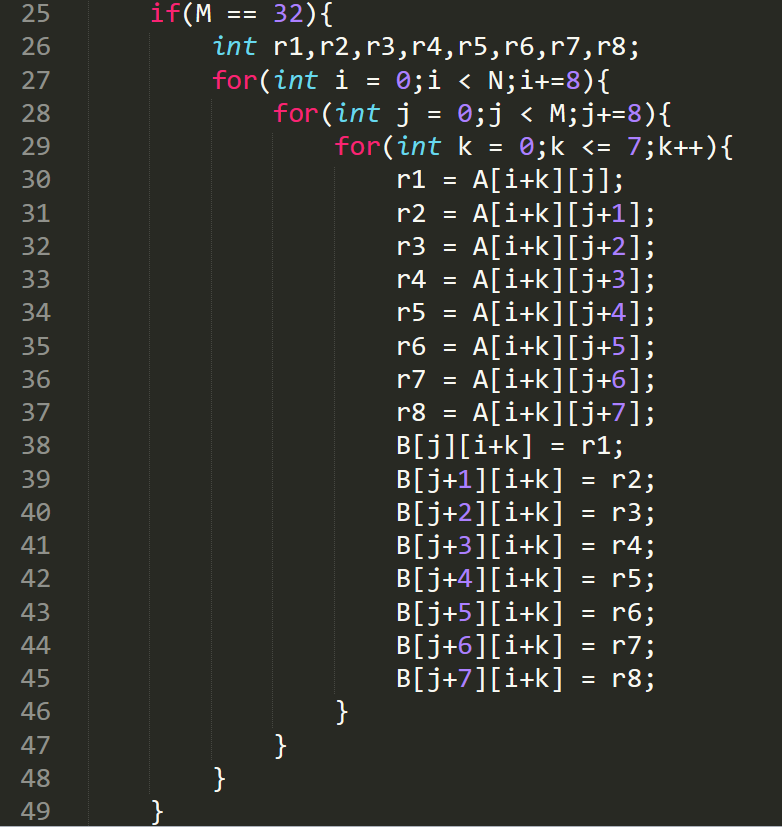


**（图10:32\*32矩阵转置优化算法缓存分析）**

**说明：局部变量使用r1-r8标记（命名取自其功能与寄存器相似），局部变量格子的底色与即将存入的元素所在缓存行对应的颜色一致。局部变量在矩阵A处按一整行/列读取，错开是为了能够看出行/列原本的底色，下同。**

A的分块每一缓存行正好对应8个元素，而剩余局部变量也有8个，正好可以全部利用。（事实上，即使循环变量数量小于4个，用于缓存的局部变量也最多只能使用8个。因为缓存结构限制，一个缓存行只能存放8个元素，而在本问题中，一般不会去缓存的额外的缓存行中的个别数据。12个局部变量的限制中，因为使用了分块，所以必须分出2个变量来循环，剩余10个变量只能缓存一个整缓存行和一个1/4缓存行，而1/4缓存行对分块处理来说意义不大，所以事实上有用的局部变量只有8个，且是可以用到的最大值。）这8个局部变量的作用其实很类似于寄存器，将极其常用的数据缓存下来。

代码如下：



**（图11:32\*32矩阵转置代码）**

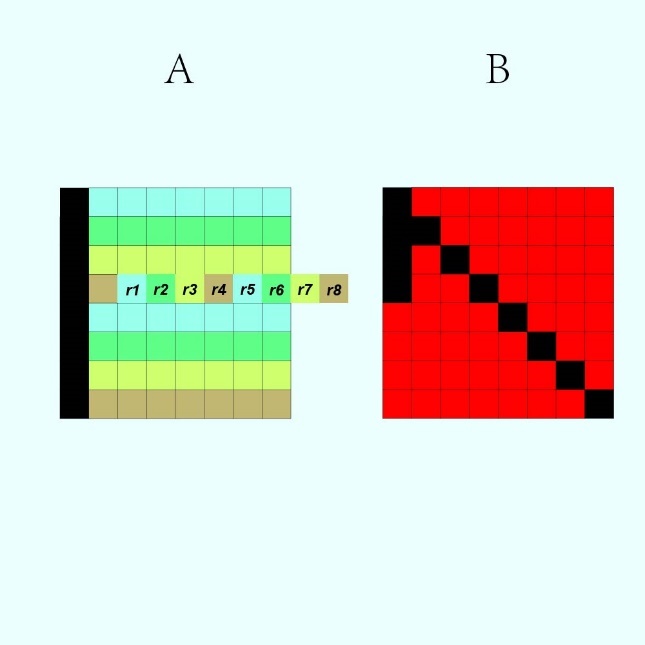
运行验证，未命中次数为287，结果通过。

**（任务B实验结果请统一参考第五部分实验结果）**

**e.64\*64分析**

对于该矩阵，如果直接实施8\*8分块，将会得到4603次不命中（使用优化算法），与同样采取该方式的d节32\*32矩阵相比，增加了许多不命中，先分析这个现象。

由于相对于上一个矩阵，64\*64矩阵每行分块数增加了一倍，这将导致缓存映射的重新分布。整个缓存能够缓存的矩阵行数，从8行减少至了4行。如果仍然采用8\*8分块，那么在块内，第(i+4)行将与第i行产生冲突不命中(i = 1,2,3,4)，这大幅增加了不命中次数。

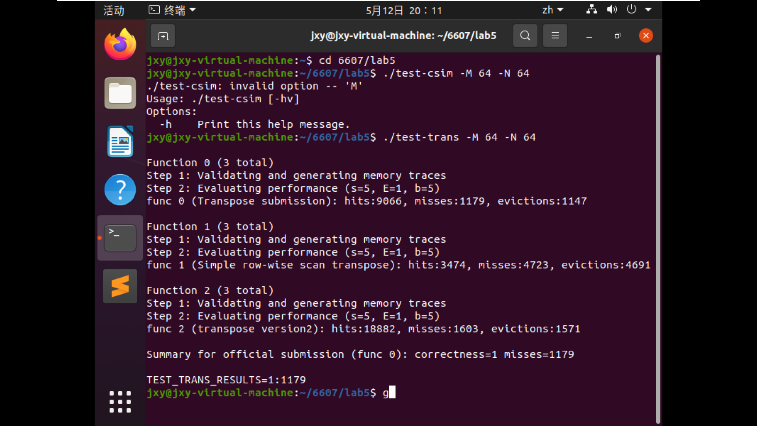


**（图12:64\*64矩阵8\*8分块转置，采用缓存整个缓存行方式）**

***说明：红色表示由于块内地址冲突导致的块驱逐。***

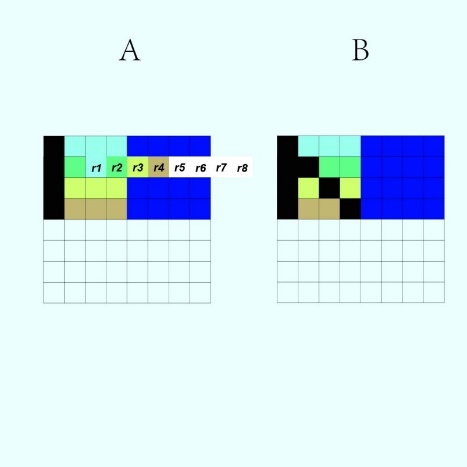
可以看出，在整个块内转置过程中，B矩阵的所有元素均为不命中，且其中几乎全部为块内冲突。对比图10，本次块内转置增加了红色的不命中，这是不命中次数较上一个矩阵大幅增加的原因。

由于64\*64矩阵每4行即产生一次冲突，我们考虑将分块改为4\*4，同样采取缓存块内一整行的方式进行转置，运行验证结果为1603次。不命中次数下降很多，但仍然未达到要求。



**（图13:64\*64矩阵4\*4分块转置结果-function2）**

事实上，由于缓存的加载方式，即使我们采用4\*4分块，高速缓存仍然会加载一行8个元素，而其中后4各元素根本未被使用就被丢弃了。在执行新的缓存块转置时，又重新加载了这一个块，产生了重复加载，进而导致不命中数量增加。



**（图14:64\*64矩阵转置4\*4分块缓存分析）**

**说明：图中蓝色为加载但未被使用即丢弃的元素，白色为未使用或不考虑。**

所以，为了充分利用缓存，我们必须使用8\*8分块，但是由于8\*8分块的内部冲突不命中，我们又需要使用4\*4分块。那么，将其结合起来，对8\*8大分块内部使用4\*4分块，尝试是否可以得到更好的效果。

考虑到对于一个分块矩阵

其对应的转置矩阵为

因此，我们可以对8\*8矩阵内部进行4\*4分块，并分别进行转置，最后将结果合并。此时，由于我们可以独立地对4个4\*4小块进行操作，仅需要在操作时考虑下一步的缓存，相对来说设计就比较轻松。

首先，对于A分块，必然要利用局部变量读取一行（否则就没有两次分块的意义了）。对于前4行，每一行的前4列可以直接进行原地转置，后4列则需要根据后续步骤进行设计。

对于A的后4行，在最终结果中，每一行的前4列将会转移至B分块的右上角小分块，而刚才缓存的后4列，则会转移至B分块的左下角。如果能在这一步更新时，就将两个小分块全部就绪。那么最后一个小分块，就可以独自执行原地转置了。

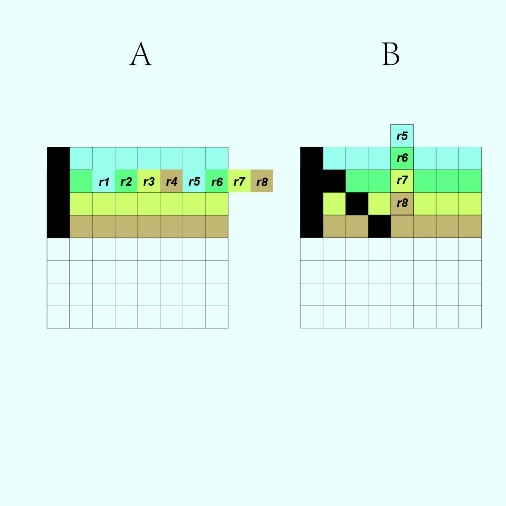
对于A右上角的小块，显然不能将其直接缓存在局部变量中，所以需要找到一个空间将其存储下来，以备后用。注意到B的右上角直到A扫描至左下角时才需要更新，B的右上角这一个小块就是一个合适的缓存区（且由于在同一行，有了左上角的转置过程，访问右上角一定是缓存命中的）。只需要在后续更新这一区域时，再按照顺序将其依次保存至目标区域中（此时目标区域B的左下角一定为空，而且由于顺序扫描，一定是按照行或者列的顺序逐一更新的，每次更新最多需要缓存4个元素值）。

最后将右下角的小块原地转置即可。

由于中间的转移过程需要数组元素的跨分块移动，因此，在B分块的右上角缓存时，可以先行将分块内部进行转置，这样在后续移动时，仅需要进行平移即可，减少涉及到的缓存行数量，降低引起不命中的可能。

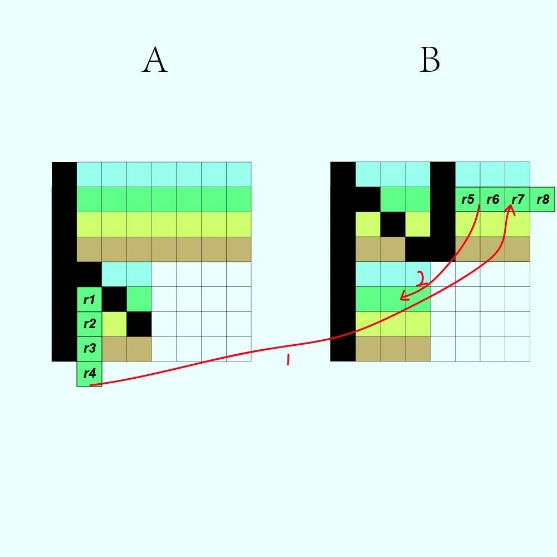
各步图示以及对应缓存状态如下。

第一步：



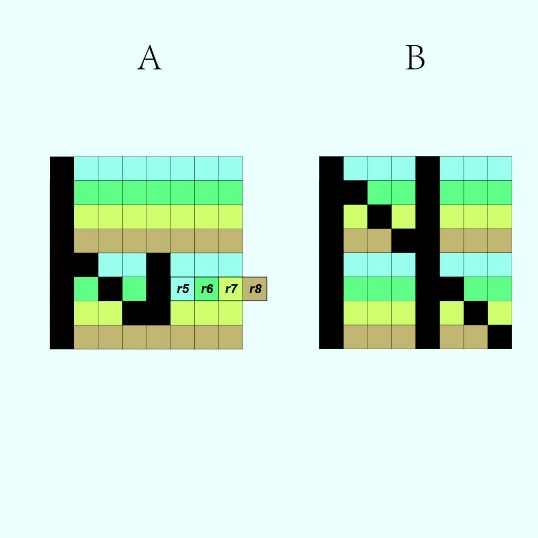
**（图15：64\*64矩阵两次分块转置第一步缓存情况-左上角原地转置和右上角缓存）**

第二步：



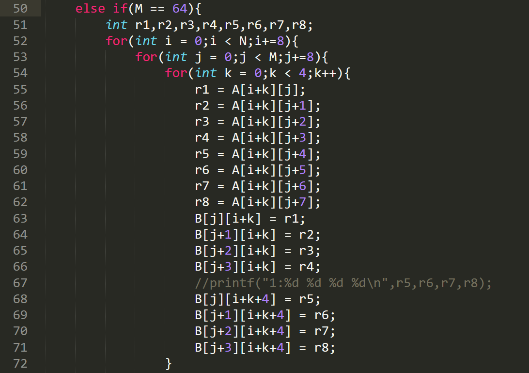
**（图16:64\*64矩阵两次分块转置第二步缓存情况-右上角与左下角元素更新）**

第三步：

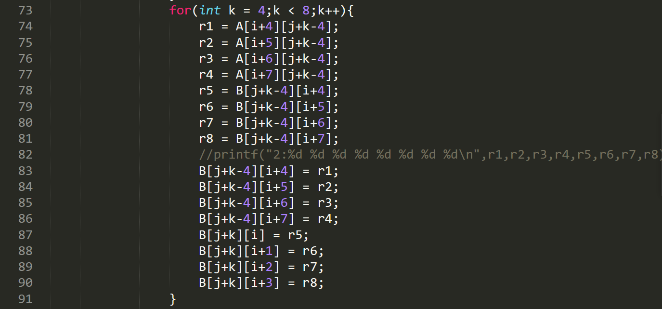


**（图17:64\*64矩阵两次分块转置第三步缓存情况-右下角原地转置）**

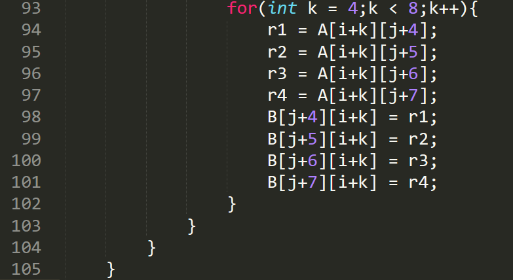
代码如下：



**（图18-1:64\*64矩阵转置代码）**



**（图18-2:64\*64矩阵转置代码）**



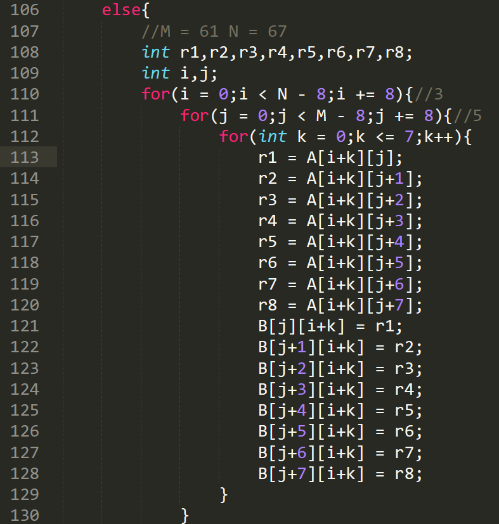
**（图18-3:64\*64矩阵转置代码）**

运行验证，不命中次数1179次，满足要求。

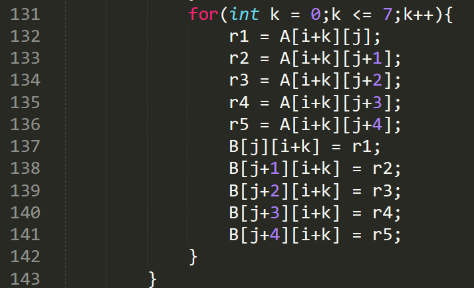
**f.61\*67分析**

对于这个矩阵，由于61和67均为质数，所以发生冲突不命中的概率大大降低，但仍然需要进行分块。我们尝试进行8\*8分块，块内按照d节所述方法进行转置。对于行列剩余的尾数部分，我们直接进行整体处理即可。

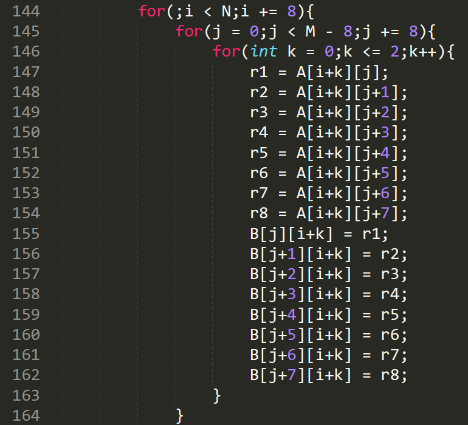
代码如下：



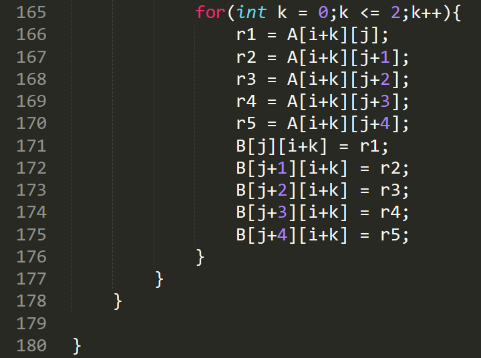
**（图19-1:61\*67矩阵转置代码）**



**（图19-2:61\*67矩阵转置代码）**



**（图19-3:61\*67矩阵转置代码）**



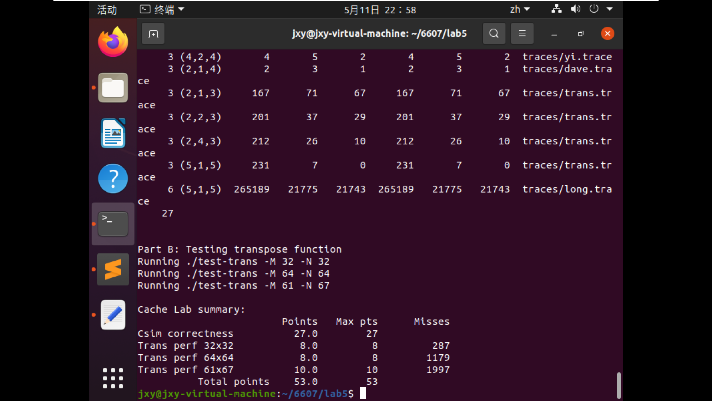
**（图19-4:61\*67矩阵转置代码）**

经过验证，8\*8分块方案不命中数为1997，满足要求。

事实上，由于这个矩阵的元素并不满足在缓存行内的对齐（即矩阵内相同列坐标对应的缓存行分布不随行坐标改变而改变），所以这样的8\*8分块在读取时会出现跨缓存行读取的情况，会造成不命中数的小幅提升。不过相对于冲突不命中带来的增幅，这个小增幅对不命中数的影响相对较小，通过运行结果也可以验证，在不做进一步优化的情况下，这种方案也能得到不错的缓存性能。

如果仔细分析转置过程，理论上应该可以将不命中次数进一步缩减。但是，越精细的优化代码量越大，且出现错误可能越大，本实验将不再做进一步的优化。

1. 实验结果：



**（图20：运行通过）**

1. 心得：

在这次实验中，我手动模拟了一个高速缓存的存储结构和读写过程，深刻理解了高速缓存的工作原理。同时，利用高速缓存模拟器和内存跟踪程序，尝试编写缓存友好的矩阵转置函数，测试了不同算法转置过程中对内存的影响，进而理解了优化缓存访问方式的重要性。在这次实验中，我学会了高速缓存的结构，和利用高速缓存提升程序性能的基本原理和思想，并尝试将其运用至实际编码过程中。

1. 参考资料。

[1]兰德尔 E.布莱恩特（Randal E. Bryant）,大卫 R.奥哈拉伦（David R. O’Hallaron）.深入理解计算机系统:第3版[M].北京:机械工业出版社,2020:35-41