15-213/18-213, Fall 2012
Cache Lab: Understanding Cache Memories
Assigned: Tuesday, October 2, 2012
Due: Thursday, October 11, 11:59PM
Last Possible Time to Turn in: Sunday, October 14, 11:59PM

1 Logistics

This is an individual project. You must run this lab on a 64-bit x86-64 machine.

> **SITE-SPECIFIC: Insert any other logistical items here, such as how to ask for help.**

2 Overview

This lab will help you understand the impact that cache memories can have on the performance of your C programs.

The lab consists of two parts. In the first part you will write a small C program (about 200-300 lines) that simulates the behavior of a cache memory. In the second part, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

3 Downloading the assignment

> **SITE-SPECIFIC: Insert a paragraph here that explains how the instructor will hand out the `cachelab-handout.tar` file to the students.**

Start by copying `cachelab-handout.tar` to a protected Linux directory in which you plan to do your work. Then give the command

```
linux> tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files. You will be modifying two files: `csim.c` and `trans.c`. To compile these files, type:

15-213/18-213,2012 年秋季 缓存实验室：了解缓存内存 分配时间：2012 年 10 月 2 日星期二 截止日期：10 月 11 日星期四晚上 11：59 最后上交时间：10 月 14 日星期日晚上 11：59

1 后勤

这是一个单独的项目。必须在 64 位 x86-64 计算机上运行此实验室。

特定于站点：在此处插入任何其他后勤项目，例如如何寻求帮助。

2 概述

本实验将帮助您了解缓存内存对 C 程序性能的影响。
该实验室由两部分组成。在第一部分中，您将编写一个小型 C 程序（大约 200-300 行）来模拟缓存的行为。在第二部分中，您将优化一个小的矩阵转置函数，目标是最大限度地减少缓存未命中次数。

3 下载作业 特定于站点：在此处插入一段，解释教师将如何向学生分发 cachelab-handout.tar 文件。

首先将 cachelab-handout.tar 复制到您计划在其中执行工作的受保护 Linux 目录。然后给出命令

```
linux> tar xvf cachelab-handout.tar
```

这将创建一个名为 cachelab-handout 的目录，其中包含许多文件。您将修改两个文件：csim.c 和 trans.c。若要编译这些文件，请键入：

```
linux> make clean
linux> make
```

WARNING: Do not let the Windows WinZip program open up your `.tar` file (many Web browsers are set to do this automatically). Instead, save the file to your Linux directory and use the Linux `tar` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files. Doing so can cause loss of data (and important work!).

4 Description

The lab has two parts. In Part A you will implement a cache simulator. In Part B you will write a matrix transpose function that is optimized for cache performance.

4.1 Reference Trace Files

The `traces` subdirectory of the handout directory contains a collection of *reference trace files* that we will use to evaluate the correctness of the cache simulator you write in Part A. The trace files are generated by a Linux program called `valgrind`. For example, typing

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

on the command line runs the executable program "`ls -l`", captures a trace of each of its memory accesses in the order they occur, and prints them on `stdout`.

`Valgrind` memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is

```
[space]operation address,size
```

The *operation* field denotes the type of memory access: "I" denotes an instruction load, "L" a data load, "S" a data store, and "M" a data modify (i.e., a data load followed by a data store). There is never a space before each "I". There is always a space before each "M", "L", and "S". The *address* field specifies a 64-bit hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation.

4.2 Part A: Writing a Cache Simulator

In Part A you will write a cache simulator in `csim.c` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

```
Linux> 使 CLEAN
Linux>使
```

类型： 警告： 不要让 Windows WinZip 程序打开您的 .tar 文件（许多 Web 浏览器设置为自动执行此操作）。相反，请将文件保存到 Linux 目录，并使用 Linux tar 程序提取文件。一般来说，对于这个类，你不应该使用Linux 以外的任何平台来修改你的文件。这样做可能会导致数据丢失（以及重要的工作！

4 描述

该实验室分为两部分。在 A 部分中，您将实现一个缓存模拟器。在 B 部分中，您将编写一个针对缓存性能进行优化的矩阵转置函数。

4.1 引用跟踪文件

讲义目录的 traces 子目录包含一组引用跟踪文件，我们将使用这些文件来评估您在 A 部分 中编写的缓存模拟器的正确性。跟踪文件由名为 valgrind 的 Linux 程序生成。例如，键入

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

在命令行上运行可执行程序“ls -l”，按其发生的顺序捕获其每个内存访问的跟踪，并将它们打印在 stdout 上。

Valgrind 内存跟踪具有以下形式：

```
我 0400d7d4,8
 货号 0421c7f0,4
 L 04f6b868,8
 编号 7ff0005c8,8
```

每行表示一次或两次内存访问。每行的格式是

【空格】操作 地址、大小

操作字段表示内存访问的类型：“I”表示指令加载，“L”表示数据加载，“S”表示数据存储，“M”表示数据修改（即数据加载后跟数据存储）。每个“我”之前从来没有空格。每个“M”、“L”和“S”之前总是有一个空格。address 字段指定 64 位十六进制内存地址。size 字段指定操作访问的字节数。

4.2 A 部分：编写缓存模拟器

在 A 部分 中，您将在 csim.c 中编写一个缓存模拟器，该模拟器将 valgrind 内存跟踪作为输入，模拟此跟踪上缓存的命中/未命中行为，并输出命中、未命中和逐出的总数。

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

The command-line arguments are based on the notation (s, E, and b) from page 597 of the CS:APP2e textbook. For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Your job for Part A is to fill in the `csim.c` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules for Part A

- Include your name and loginID in the header comment for `csim.c`.

我们为您提供了引用缓存模拟器的二进制可执行文件，称为 csim-ref，该模拟器模拟 valgrind 跟踪文件上具有任意大小和关联性的缓存的行为。在选择要逐出的缓存行时，它使用 LRU（最近最少使用）替换策略。

引用模拟器采用以下命令行参数：

用法：./csim-ref [-hv] -s -~~E -b <b> -t~~

- -h：打印使用信息的可选帮助标志
- -v：显示跟踪信息的可选详细标志
- -s ~~： 设置索引位数（S = 2是设置数）~~
- -E ： 关联性（每组行数）
- -b <b>： 块位数（B = 2 是块大小）
- -t ：要重放的 valgrind 跟踪的名称

命令行参数基于 CS：APP2e 教科书第 597 页的符号（s、E 和 b）。例如：

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace 命中数：4
未命中数：5 逐出数：3
```

详细模式下的相同示例：

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace L 10.1 未命
中 M 20.1 未命中 L 22.1 命中 S 18.1 命中
```

```
L 110,1 未被驱逐
L 210,1 未被驱逐
M 12,1 未被驱逐命中
命中数：4 未命中数：5 驱逐数：3
```

第 A 部分的工作是填写 csim.c 文件，以便它采用相同的命令行参数并生成与参考模拟器相同的输出。请注意，此文件几乎完全为空。
您需要从头开始编写它。

A 部分的编程规则

- 在 csim.c 的标题注释中包含您的姓名和 loginID 。

- Your `csim.c` file must compile without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function. Type "man malloc" for information about this function.
- For this lab, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `valgrind` always puts "I" in the first column (with no preceding space), and "M", "L", and "S" in the second column (with a preceding space). This may help you parse the trace.
- To receive credit for Part A, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hit_count, miss_count, eviction_count);
```

- For this this lab, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.

4.3 Part B: Optimizing Matrix Transpose

In Part B you will write a transpose function in `trans.c` that causes as few cache misses as possible.

Let A denote a matrix, and A_{ij} denote the component on the ith row and jth column. The *transpose* of A, denoted A^T, is a matrix such that $A_{ij} = A^T_{ji}$.

To help you get started, we have given you an example transpose function in `trans.c` that computes the transpose of $N \times M$ matrix A and stores the results in $M \times N$ matrix B:

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
```

The example transpose function is correct, but it is inefficient because the access pattern results in relatively many cache misses.

Your job in Part B is to write a similar function, called `transpose_submit`, that minimizes the number of cache misses across different sized matrices:

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

Do *not* change the description string ("`Transpose submission`") for your `transpose_submit` function. The autograder searches for this string to determine which transpose function to evaluate for credit.

- 您的 csim.c 文件必须在没有警告的情况下编译才能获得积分。
- 模拟器必须对任意 s、E 和 b 正常工作。这意味着您需要使用 malloc 函数为模拟器的数据结构分配存储。键入“man malloc”
  以获取有关此功能的信息。
- 在本实验中，我们只对数据缓存性能感兴趣，因此模拟器应忽略所有指令缓存访问（以“I”开头的行）。回想一下，valgrind 总是将“I”放在第一列（前面没有空格），将“M”、“L”和“S”放在第二列（前面有空格）。这可以帮助您分析跟踪。
- 要获得 A 部分的积分，您必须在主函数末尾调用函数 printSummary ，其中包含命中、未命中和逐出的总数：

  ```
  printSummary （hit_count， miss_count， eviction_count ）；
  ```

- 在本实验中，您应假定内存访问正确对齐，以便单个内存访问永远不会跨越块边界。通过做出此假设，您可以忽略 valgrind 跟踪中的请求大小。

4.3 B 部分：优化矩阵转置

在 B 部分中，您将在 trans.c 中编写一个转置函数，该函数会导致尽可能少的缓存未命中。

设 A 表示矩阵，Adenote 表示第 i 行和第 j 列上的分量。A 的转置，表示为 A，是一个矩阵，使得 A= A。

为了帮助您入门，我们在 trans.c 中为您提供了一个转置函数示例，该函数计算 N × M 矩阵 A 的转置，并将结果存储在 M × N 矩阵 B 中：

```
char trans_desc [] = “简单逐行扫描转置”;void trans（int M， int
N， int A[N][M]， int B[M][N]）
```

示例转置函数是正确的，但它效率低下，因为访问模式会导致相对较多的缓存未命中。

您在 B 部分中的工作是编写一个类似的函数，称为 transpose_submit ，该函数可以最大程度地减少不同大小矩阵中的缓存未命中次数：

```
char transpose_submit_desc [] = “转置提交”;void transpose_submit （int M，
int N， int A[N][M]， int B[M][N]）；
```

请勿更改 transpose_submit 函数的描述字符串（“转置提交”）。自动评分器搜索此字符串以确定要评估信用的转置函数。

Programming Rules for Part B

- Include your name and loginID in the header comment for `trans.c`.
- Your code in `trans.c` must compile without warnings to receive credit.
- You are allowed to define at most 12 local variables of type `int` per transpose function.[1]
- You are not allowed to side-step the previous rule by using any variables of type `long` or by using any bit tricks to store more than one value to a single variable.
- Your transpose function may not use recursion.
- If you choose to use helper functions, you may not have more than 12 local variables on the stack at a time between your helper functions and your top level transpose function. For example, if your transpose declares 8 variables, and then you call a function which uses 4 variables, which calls another function which uses 2, you will have 14 variables on the stack, and you will be in violation of the rule.
- Your transpose function may not modify array A. You may, however, do whatever you want with the contents of array B.
- You are NOT allowed to define any arrays in your code or to use any variant of `malloc`.

5 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 60 points:

- Part A: 27 Points
- Part B: 26 Points
- Style: 7 Points

5.1 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters and traces. There are eight test cases, each worth 3 points, except for the last case, which is worth 6 points:

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
```

[1] The reason for this restriction is that our testing code is not able to count references to the stack. We want you to limit your references to the stack and focus on the access patterns of the source and destination arrays.

B 部分的编程规则

- 在 trans.c 的标题注释中包含您的姓名和登录 ID。
- trans.c 中的代码必须在没有警告的情况下编译才能获得信用。
- 每个转置函数最多可以定义 12 个 int 类型的局部变量。
- 不允许通过使用任何 long 类型的变量或使用任何位技巧将多个值存储到单个变量来回避上一条规则。
- 转置函数可能不使用递归。
- 如果选择使用帮助程序函数，则在帮助程序函数和顶级转置函数之间，堆栈上的局部变量一次不得超过 12 个。例如，如果转置声明了 8 个变量，然后调用一个使用 4 个变量的函数，该函数调用另一个使用 2 的函数，则堆栈上将有 14 个变量，并且将违反规则。
- 转置函数不能修改数组 A。但是，您可以对数组 B 的内容执行任何操作。
- 不允许在代码中定义任何数组或使用 malloc 的任何变体。

5 评估

本节介绍如何评估您的工作。本实验的满分为 60 分：

- A 部分：27 分
- B 部分：26 分
- 风格： 7 Points

5.1 A 部分的评估

对于 A 部分，我们将使用不同的缓存参数和跟踪来运行缓存模拟器。有 8 个测试用例，每个测试用例值 3 分，但最后一个用例值 6 分：

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace linux> ./csim
-s 4 -E 2 -b 4 -t traces/yi.trace linux> ./csim -s 2 -E 1 -b 4
-t traces/dave.trace linux> ./csim -s 2 -E 1 -b 3 -t
traces/trans.trace linux> ./csim -s 2 -E 2 -b 3 -t
traces/trans.trace
```

此限制的原因是我们的测试代码无法计算对堆栈的引用。我们希望您限制对堆栈的引用，并专注于源阵列和目标阵列的访问模式。

```
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

For each test case, outputting the correct number of cache hits, misses and evictions will give you full credit for that test case. Each of your reported number of hits, misses and evictions is worth 1/3 of the credit for that test case. That is, if a particular test case is worth 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points.

5.2 Evaluation for Part B

For Part B, we will evaluate the correctness and performance of your `transpose_submit` function on three different-sized output matrices:

- 32×32 ($M = 32$, $N = 32$)
- 64×64 ($M = 64$, $N = 64$)
- 61×67 ($M = 61$, $N = 67$)

5.2.1 Performance (26 pts)

For each matrix size, the performance of your `transpose_submit` function is evaluated by using `valgrind` to extract the address trace for your function, and then using the reference simulator to replay this trace on a cache with parameters ($s = 5$, $E = 1$, $b = 5$).

Your performance score for each matrix size scales linearly with the number of misses, m, up to some threshold:

- 32×32: 8 points if $m < 300$, 0 points if $m > 600$
- 64×64: 8 points if $m < 1,300$, 0 points if $m > 2,000$
- 61×67: 10 points if $m < 2,000$, 0 points if $m > 3,000$

Your code must be correct to receive any performance points for a particular size. Your code only needs to be correct for these three cases and you can optimize it specifically for these three cases. In particular, it is perfectly OK for your function to explicitly check for the input sizes and implement separate code optimized for each case.

```
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace linux>
./csim -s 5 -E 1 -b 5 -t traces/trans.trace linux> ./csim -s 5
-E 1 -b 5 -t traces/long.trace
```

您可以使用参考模拟器 csim-ref 来获取每个测试用例的正确答案。
在调试期间，使用 -v 选项详细记录每次命中和未命中。

对于每个测试用例，输出正确数量的缓存命中、未命中和逐出将为您提供该测试用例的全部功劳。您报告的每个命中数、未命中数和逐出数都相当于该测试用例的 1/3 功劳。也就是说，如果一个特定的测试用例值 3 分，并且您的模拟器输出了正确的命中数和未命中数，但报告了错误的逐出次数，那么您将获得 2 分。

5.2 B 部分的评估

对于 B 部分，我们将评估 transpose_submit 函数在三个不同大小的输出矩阵上的正确性和性能：

- 32×32 （M = 32， N = 32）
- 64×64 （M = 64， N = 64）
- 61×67 （M = 61， N = 67）

5.2.1 表演 （26 分）

对于每个矩阵大小，使用 valgrind 提取函数的地址跟踪，然后使用引用模拟器在具有参数 （s = 5， E = 1， b = 5） 的缓存上重播此跟踪，来评估 transpose_submit 函数的性能。
每个矩阵大小的性能分数与未命中数 m 成线性关系，直至达到某个阈值：

- 32×32: 如果 m < 300，则得 8 分，如果 m > 600，则得 0 分
- 64×64: 如果 m < 1， 300，则为 8 分，如果 m > 2， 000，则为 0 分
- 61×67: 如果 m < 2， 000，则为 10 分，如果 m > 3， 000，则为 0 分

您的代码必须正确，才能获得特定大小的任何性能点。您的代码只需要针对这三种情况是正确的，您可以专门针对这三种情况对其进行优化。特别是，您的函数完全可以显式检查输入大小并实现针对每种情况优化的单独代码。

5.3 Evaluation for Style

There are 7 points for coding style. These will be assigned manually by the course staff. Style guidelines can be found on the course website.

The course staff will inspect your code in Part B for illegal arrays and excessive local variables.

6 Working on the Lab

6.1 Working on Part A

We have provided you with an autograding program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
linux> make
linux> ./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
```

For each test, it shows the number of points you earned, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference simulator.

Here are some hints and suggestions for working on Part A:

- Do your initial debugging on the small traces, such as `traces/dave.trace`.
- The reference simulator takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. You are not required to implement this feature in your `csim.c` code, but we strongly recommend that you do so. It will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.
- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

```
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
```

5.3 风格评估

编码风格有 7 点。这些将由课程工作人员手动分配。风格指南可以在课程网站上找到。

课程工作人员将检查您在 B 部分中的代码是否存在非法数组和过多的局部变量。

6 在实验室工作

6.1 处理 A 部分

我们为您提供了一个名为 test-csim 的自动分级程序，用于测试缓存模拟器在引用跟踪上的正确性。请务必在运行测试之前编译模拟器：

```
Linux>使
linux> ./test-csim
您的模拟器 参考模拟器 点数 （s，E，b） 命中 未命中
逐出 命中未命中 逐出
      3 （1,1,1） 9 8 6 9 8 6 迹线/yi2.trace
      3 （4,2,4） 4 5 2 4 5 2 迹线/yi.trace
      3 （2,1,4） 2 3 1 2 3 1 跟踪/dave.trace
      3 （2,1,3） 167 71 67 167 71 67 迹线/trans.trace
      3 （2,2,3） 201 37 29 201 37 29 迹线/trans.trace
      3 （2,4,3） 212 26 10 212 26 10 迹线/trans.trace
      6 （5,1,5） 265189 21775 21743 265189 21775 21743 traces/long.trace
     27
```

对于每个测试，它都会显示您获得的分数、缓存参数、输入跟踪文件，以及模拟器和参考模拟器的结果比较。

以下是处理 A 部分的一些提示和建议：

- 对小跟踪（例如 traces/dave.trace）进行初始调试。
- 引用模拟器采用可选的 -v 参数，该参数启用详细输出，显示每次内存访问导致的命中、未命中和逐出。不需要在 csim.c 代码中实现此功能，但我们强烈建议您这样做。它允许你直接将模拟器的行为与引用跟踪文件上的引用模拟器进行比较，从而帮助你进行调试。

- 我们建议您使用 getopt 函数来解析命令行参数。您将需要以下头文件：

```
#include < getopt.h>
#include < stdlib.h>
#include < unistd.h>
```

See "`man 3 getopt`" for details.

- Each data load (L) or store (S) operation can cause at most one cache miss. The data modify operation (M) is treated as a load followed by a store to the same address. Thus, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.
- If you would like to use C0-style contracts from 15-122, you can include `contracts.h`, which we have provided in the handout directory for your convenience.

6.2 Working on Part B

We have provided you with an autograding program, called `test-trans.c`, that tests the correctness and performance of each of the transpose functions that you have registered with the autograder.

You can register up to 100 versions of the transpose function in your `trans.c` file. Each transpose version has the following form:

```
/* Header comment */
char trans_simple_desc[] = "A simple transpose";
void trans_simple(int M, int N, int A[N][M], int B[M][N])
{
    /* your transpose code here */
}
```

Register a particular transpose function with the autograder by making a call of the form:

```
registerTransFunction(trans_simple, trans_simple_desc);
```

in the `registerFunctions` routine in `trans.c`. At runtime, the autograder will evaluate each registered transpose function and print the results. Of course, one of the registered functions must be the `transpose_submit` function that you are submitting for credit:

```
registerTransFunction(transpose_submit, transpose_submit_desc);
```

See the default `trans.c` function for an example of how this works.

The autograder takes the matrix size as input. It uses `valgrind` to generate a trace of each registered transpose function. It then evaluates each trace by running the reference simulator on a cache with parameters ($s = 5$, $E = 1$, $b = 5$).

For example, to test your registered transpose functions on a 32×32 matrix, rebuild `test-trans`, and then run it with the appropriate values for M and N:

```
linux> make
linux> ./test-trans -M 32 -N 32
Step 1: Evaluating registered transpose funcs for correctness:
func 0 (Transpose submission): correctness: 1
```

有关详细信息，请参见“man 3 getopt ”。

- 每个数据加载 （L） 或存储 （S） 操作最多会导致一次缓存未命中。数据修改操作 （M） 被视为加载，然后存储到同一地址。因此，M 操作可能导致两次缓存命中，或者一次未命中和一次命中，以及可能的逐出。
- 如果您想使用 15-122 的 C0 样式合约，您可以包含 contracts.h，为了方便起见，我们在讲义目录中提供了它。

6.2 处理 B 部分

我们为您提供了一个名为 test-trans.c 的自动分级程序，用于测试您在自动分级器中注册的每个转置函数的正确性和性能。
您可以在 trans.c 文件中注册最多 100 个版本的 transpose 函数。每个转置版本都具有以下形式：

```
/* 标题注释 */ char trans_simple_desc [] = “一个简单的转置”;void
trans_simple （int M， int N， int A[N][M]， int B[M][N]） {

/* 你的转置代码在这里 */ }
```

通过调用以下表单，向自动评分器注册特定的转置函数：

```
registerTransFunction （trans_simple ， trans_simple_desc ）;
```

在 trans.c 的 registerFunctions 例程中。在运行时，自动评分器将评估每个注册的转置函数并打印结果。当然，其中一个注册函数必须是您提交积分的transpose_submit 函数：

```
registerTransFunction （transpose_submit ， transpose_submit_desc ）;
```

有关其工作原理的示例，请参阅默认的 trans.c 函数。

自动分级器将矩阵大小作为输入。它使用 valgrind 生成每个已注册转置函数的跟踪。然后，它通过在缓存上运行具有参数 （s = 5， E = 1， b = 5） 的引用模拟器来评估每条跟踪。
例如，要在 32 × 32 矩阵上测试已注册的转置函数，请重新生成 test-trans，然后使用 M 和 N 的相应值运行它：

```
linux> make linux> ./test-trans -M 32 -N 32 第 1 步：评估已注册的转置函数的正确
性： func 0（转置提交）： 正确性：1
```

```
func 1 (Simple row-wise scan transpose): correctness: 1
func 2 (column-wise scan transpose): correctness: 1
func 3 (using a zig-zag access pattern): correctness: 1

Step 2: Generating memory traces for registered transpose funcs.

Step 3: Evaluating performance of registered transpose funcs (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151
func 2 (column-wise scan transpose): hits:870, misses:1183, evictions:1151
func 3 (using a zig-zag access pattern): hits:1076, misses:977, evictions:945

Summary for official submission (func 0): correctness=1 misses=287
```

In this example, we have registered four different transpose functions in `trans.c`. The `test-trans` program tests each of the registered functions, displays the results for each, and extracts the results for the official submission.

Here are some hints and suggestions for working on Part B.

- The `test-trans` program saves the trace for function i in file `trace.f`i.[2] These trace files are invaluable debugging tools that can help you understand exactly where the hits and misses for each transpose function are coming from. To debug a particular function, simply run its trace through the reference simulator with the verbose option:

  ```
  linux> ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f0
  S 68312c,1 miss
  L 683140,8 miss
  L 683124,4 hit
  L 683120,4 hit
  L 603124,4 miss eviction
  S 6431a0,4 miss
  ...
  ```

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem. Think about the potential for conflict misses in your code, especially along the diagonal. Try to think of access patterns that will decrease the number of these conflict misses.

- Blocking is a useful technique for reducing cache misses. See

  `http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf`

  for more information.

[2] Because `valgrind` introduces many stack accesses that have nothing to do with your code, we have filtered out all stack accesses from the trace. This is why we have banned local arrays and placed limits on the number of local variables.

/test-trans -M 32 -N 32 步骤 1：评估已注册的转置函数的正确性： func 0（转置提交）： 正确性：1 func 1（简单行扫描转置）： 正确性：1 func 2（按列扫描转置）： 正确性：1 func 3（使用之字形访问模式）： 正确性：1

第 2 步：为已注册的转置函数生成内存跟踪。

步骤 3：评估已注册转置函数的性能 （s=5， E=1， b=5） func 0（转置提交）： 命中数：1766， 未命中数：287， 逐出：255 func 1（简单逐行扫描转置）： 命中数：870， 未命中数：1183， 逐出：1151 func 2（按列扫描转置）： 命中数：870， 未命中数：1183， 逐出：1151 func 3（使用之字形访问模式）： 命中数：1076， 未命中：977， 逐出：945

正式提交摘要 （func 0）：正确性=1 未命中=287

在此示例中，我们在 trans.c 中注册了四个不同的转置函数。test-trans 程序测试每个已注册的函数，显示每个函数的结果，并提取结果以供正式提交。

以下是处理 B 部分的一些提示和建议。

- test-trans 程序将函数 i 的跟踪保存在文件 trace.fi 中。这些跟踪文件是非常宝贵的调试工具，可以帮助您准确了解每个转置函数的命中和未命中来自何处。若要调试特定函数，只需使用 verbose 选项通过引用模拟器运行其跟踪：

  ```
  linux> ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f0 S 68312c，
  1 miss L 683140,8 miss

  L 683124,4 命中
  L 683120,4 命中
  L 603124,4 驱逐失误
  S 6431a0,4 未命中
  ...
  ```

- 由于转置函数是在直接映射的缓存上评估的，因此冲突未命中是一个潜在的问题。考虑代码中可能出现的冲突遗漏，尤其是沿对角线。尝试考虑能够减少这些冲突未命中次数的访问模式。

- 阻塞是减少缓存未命中的有用技术。看

  http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf

  了解更多信息。

[2] 由于 valgrind 引入了许多与您的代码无关的堆栈访问，因此我们从跟踪中筛选掉了所有堆栈访问。这就是为什么我们禁止局部数组并限制局部变量的数量。

6.3 Putting it all Together

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator and transpose code. This is the same program your instructor uses to evaluate your handins. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` to evaluate your submitted transpose function on the three matrix sizes. Then it prints a summary of your results and the points you have earned.

To run the driver, type:

```
linux> ./driver.py
```

7 Handing in Your Work

Each time you type `make` in the `cachelab-handout` directory, the Makefile creates a tarball, called `userid-handin.tar`, that contains your current `csim.c` and `trans.c` files.

> **SITE-SPECIFIC: Insert text here that tells each student how to hand in their `userid-handin.tar` file at your school.**

IMPORTANT: Do not create the handin tarball on a Windows or Mac machine, and do not handin files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files.

6.3 把它们放在一起

我们为您提供了一个名为 ./driver.py 的驱动程序，用于对模拟器和转置代码执行完整评估。这与您的教师用来评估您的交出的程序相同。驱动程序使用 test-csim 来评估模拟器，并使用 test-trans 来评估提交的三种矩阵大小的转置函数。然后，它会打印您的结果摘要和您获得的积分。

若要运行驱动程序，请键入：

```
linux> ./driver.py
```

7 交出您的作品

每次在 cachelab-handout 目录中键入 make 时，Makefile 都会创建一个名为 userid-handin.tar 的压缩包，其中包含您当前的 csim.c 和 trans.c 文件。

> 特定于站点：在此处插入文本，告诉每个学生如何在您的学校提交他们的 useridhandin.tar 文件。

重要说明：不要在Windows或Mac计算机上创建上交压缩包，也不要上交任何其他存档格式的文件，如.zip、.gzip或.tgz文件。