15-213/18-213, 2012 年秋
高速缓存实验室：了解高速缓存内存2012
年 10 月 2 日（星期二）到期：10 月 11
日（星期四）11:59 下午
最后时间：10 月 14 日（星期日）11:59

1 物流

这是一个个人项目。您必须在 64 位 x86-64 机器上运行本实验室。

现场细节：在此插入任何其他后勤项目，例如如何寻求帮助。

2 概述

本实验室将帮助了解高速缓冲存储器对 C 程序性能的影响。

本实验由两部分组成。第一部分是编写一个小型 C 程序（约 200-300 行），模拟高速缓冲存储器的行为。第二部分中，您将优化一个小型矩阵转置函数，目标是最大限度地减少缓存丢失的次数。

3 下载作业

站点特定：在此插入一段文字，说明教师将如何向学生分发 `cachelab-handout.tar` 文件。

首先，将 `cachelab-handout.tar` 复制到受保护的 Linux 目录中，并计划在该目录中进行工作。然后执

行命令

```
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` 。要编译这两个文件，请键入

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

警告：不要让 Windows WinZip 程序打开你的 `.tar` 文件（许多网络浏览器都设置为自动打开）。相反，请将文件保存到你的 Linux 目录中，然后使用 Linux `tar` 程序解压文件。一般来说，在这门课上，你绝对不应该使用 Linux 以外的任何平台来修改你的文件。这样做会导致数据丢失（和重要工作丢失！）。

4 说明

本实验分为两部分。在 A 部分，你将实现一个高速缓存模拟器。在 B 部分中，您将编写一个针对缓存性能进行了优化的矩阵转置函数。

4.1 参考跟踪文件

讲义目录下的 `traces` 子目录包含一系列*参考*跟踪文件，我们将用它们来评估你在 A 部分中编写的缓存模拟器正确性。例如，输入

```
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`"，按顺序捕捉每次内存访问的轨迹，并打印到 `stdout` 上。

`Valgrind` 内存跟踪的形式如下：

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

每一行表示一次或两次内存访问。每行的格式为

`[`操作地址、大小

*操作*字段表示内存访问类型：I "表示指令加载，"L "数据加载，"S "表示存储，"M "表示数据修改（即数据加载后进行数据存储）。每个 "I "前都没有空格。每个 "M"、"L "和 "S "前总是有一个空格。*地址*字段指定 64 位十六进制内存地址。*大小*字段指定操作访问字节数。

4.2 A 部分：编写缓存模拟器

在 A 部分中您将在 `csim.c` 中编写一个高速缓存模拟器，将 `valgrind` 内存跟踪作为输入，在该跟踪上模拟高速缓存的命中/未命中行为，并输出命中、未命中和唤出的总次数。

我们为您提供了名为 `csim-ref` 的*引用缓存模拟器*的二进制可执行文件，它可以在 `valgrind` 跟踪文件上模拟具有任意大小和关联性的缓存的行为。在选择驱逐哪个缓存行时，它会使用 LRU（最近最少使用）替换策略。

参考模拟器需要以下命令行参数：

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

- `-h`:可选的帮助标志，可打印使用信息
- `-v`:显示跟踪信息的可选 "冗长 "标记
- `-s <s>`：集合索引位数（$S = 2^s$为集合数）
- `-E <E>`：关联性（每组的行数）
- `-b <b>`：数据块位数（$B = 2^b$为数据块大小）
- `-t <tracefile>`：要重放的 `valgrind` 跟踪文件名称

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

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

同样的示例，采用详细说明模式：

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1失误
M 20，1 次未命中 L
22，1 次命中
S 18，1 次命中
L 110,1 错过驱逐
L 210,1 错过驱逐
M 12,1 未命中驱逐命中:4 未命中:5 驱
逐:3
```

A 部分的任务是填写 `csim.c` 文件，使其使用相同的命令行参数，并产生与参考模拟器相同的输出。请注

意，该文件几乎完全空白。你需要从头开始编写。

A 部分程序设计规则

- 在 `csim.c` 的头注释中包含您的姓名和登录名。

- 您的 `csim.c` 文件在编译时必须无警告，才能获得学分。
- 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空间。输入 "man malloc "可获取该函数的相关信息。
- 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），而将 "M"、"L "和 "S "放在第二列（前面有空格）。这可能有助于您解析跟踪。
- 要获得 A 部分的学分，您必须在主函数的末尾调用 `printSummary` 函数，输入命中、未命中和驱逐的总数：

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

- 在本实验中，您应假设内存访问已正确对齐，因此单次内存访问绝不会跨越块边界。有了这个假设，你就可以忽略 `valgrind` 跟踪中的请求大小。

4.3 B 部分：优化矩阵转置

在 B 部分，您将在 `trans.c` 中编写一个转置函数，尽可能减少缓存丢失。

让 *A* 表示矩阵，A_{ij} 表示第行和第 j 列上的分量。*A* 的*转置*，表示 A^T，是一个矩阵，使得 $A_{ij}= {}^{A(T)}_{ji}$。

为了帮助您入门，我们在 `trans.c` 中提供了一个转置函数示例，它可以计算 *N*× *M* 矩阵 *A* 的转置，并将结果存储在 *M*× *N* 矩阵 *B* 中：

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

示例中的转置函数是正确的，但效率很低，因为访问模式会导致相对较多的缓存缺失。

在 B 部分中，你的任务是编写一个类似的函数，名为 `transpose_submit`，该函数可以最大限度地减少不同大小矩阵的缓存丢失次数：

```
char transpose_submit_desc[]= "Transpose submission";
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"）。自动交易程序会搜索该字符串，以确

定要对哪个转置函数进行信用评估。

B 部分程序设计规则

- 在 `trans.c` 的页眉注释中包含您的姓名和登录名。
- 您在 `trans.c` 中的代码必须在编译时不出现警告，才能获得学分。
- 每个转置函数最多允许定义 12 个 `int` 类型的局部变量[1]。
- 您使用任何 `long` 类型的变量，或使用任何位技巧将多个值存储到一个变量中，从而绕过前面的规则。
- 您的转置函数可能不使用递归。
- 如果选择使用辅助函数，那么在辅助函数和顶级转置函数之间，堆栈上局部变量不得超过 12 个。例如，如果您的 transpose 声明了 8 个变量，然后调用了一个使用 4 个变量的函数，该函数又调用了另一个使用 2 个变量的函数，那么堆栈上就会有 14 个变量，这就违反了规则。
- 但是，您可以对数组 B 的内容进行任何操作。
- 不允许在代码中定义任何数组或使用任何 `malloc` 变体。

5 评估

本部分介绍了如何评估您的作业。本实验的满分为 60 分：

- A 部分：27 分
- B 部分：26 分
  - 风格：7 分

5.1 A 部分的评估

在 A 部分中，我们将使用不同的缓存参数和轨迹运行您的缓存模拟器。共有 8 个测试用例，每个测试用例得 3 分，最后一个测试用例得 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
```

[1]之所以有此限制，是因为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，重点关注源数组和目标数组的访问模式。

```
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
```

您可以使用参考模拟器 `csim-ref` 获取每个测试用例的正确答案。在调试过程中，使用 `-v` 选项可以详细记录每次测试的成功和失败。

对于每个测试用例，输出正确的高速缓存命中、未命中和唤出次数将获得该测试用例的全部学分。您所报告的命中、未命中和删除次数各占该测试用例学分 1/3。也就是说，如果某个测试用例分值为 3 分，而您的模拟器输出了正确的命中和未命中次数，但报告了错误的驱逐次数，那么您将获得 2 分。

5.2 B 部分的评估

在 B 部分，我们将在三个不同大小的输出矩阵上评估 `transpose_submit` 函数的正确性和性能：

- 32× 32 (m= 32, n= 32)
- 64× 64 (m= 64, n= 64)
- 61× 67 (m= 61, n= 67)

5.2.1 性能（26 分）

对于每种矩阵大小，我们都会使用 `valgrind` 提取的地址轨迹，然后使用参考模拟器在缓存上重放该轨迹，并设置参数（s= 5，E= 1，b= 5），从而评估 `transpose_submit` 函数的性能。

每种矩阵大小的性能得分与未命中次数 m 成线性关系，直至某个阈值：

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

您的代码必须正确无误，才能为特定大小获得任何性能点。您的代码只需在这三种情况下正确无误，并且可以专门针对这三种情况进行优化。特别是，您的函数完全可以明确检查输入大小，并针对每种情况执行单独的优化代码。

5.3 风格评估

编码风格有 7 分。这些分数将由课程工作人员手动分配。风格指南可在课程网站上找到。

课程工作人员将在 B 部分检查您的代码是否存在非法数组和过多局部变量。

6 在实验室工作

6.1 A 部分的工作

我们为您提供了一个名为 `test-csim` 的自动交易程序，用于在参考轨迹上测试高速缓存模拟器正确性。运行测试前，请务必编译您的模拟器：

```
linux> make
linux> ./test-csim
                        您的模拟器                参考模拟器
积分  (s,E,b)     点击数     小姐     驱逐     点击数     小姐     驱逐
   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
```

对于每个测试，它都会显示你获得的分数、缓存参数、输入跟踪文件，以及你的模拟器和参考模拟器的结果对比。

以下是一些关于 A 部分工作的提示和建议：

- 在小跟踪（如 `traces/dave.trace`）上进行初始调试。
- 引用模拟器使用一个可选的 `-v` 参数来启用冗长输出，显示每次内存访问的命中、未命中和唤出情况。您不必在 `csim.c` 代码中实现这一功能，但我们强烈建议您这样做。您可以直接比较您的模拟器与参考跟踪文件上的参考模拟器的行为，从而帮助您进行调试。
- 我们您使用 `getopt` 函数来解析命令行参数。您需要以下头文件：

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

详见 "man 3 getopt"。

- 每次数据加载 (L) 或存储 (S) 操作最多只能导致一次缓存缺失。数据修改操作 (M)操作被视为加载操作，随后是存储到同一地址的操作。因此，一个 M 操作可能会导致两次高速缓存命中，或一次未命中和一次命中加上一次可能的驱逐。
- 如果您想使用 15-122 中 C0 风格合同，可以在讲义目录中加入 contracts.h，以方便使用。

6.2 B 部分的工作

我们为您提供了一个名为 test-trans.c 的自动交易程序，用于测试您在自动交易软件中注册的每个转置函数的正确性和性能。

您可以在 trans.c 文件中注册多达 100 个版本的转置函数。每个转置版本的形式如下

```
/* 页眉注释 */
char trans_simple_desc[]= "简单转置";
void trans_simple(int M, int N, int A[N][M], int B[M][N])
{
    /* 在此转置代码 */
}
```

通过调用以下形式，在自动交易系统中注册一个特定的转置函数：

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

在 trans.c 中的 registerFunctions 例程中。运行时，自动跟踪器将评估每个经过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要提交的 transpose_submit 函数：

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

请参阅默认的 trans.c 函数，了解其工作原理。

自动跟踪器将矩阵大小作为输入。它使用 valgrind 生成每个已注册变换函数的轨迹。然后，它通过在缓存上运行参考模拟器来评估每个跟踪，缓存参数为（s= 5，E= 1，b= 5）。

例如，要在 32× 32 矩阵上测试已注册的转置函数，可重建 test-trans，然后使用适当的 M 和 N 值运行它：

```
linux> make
```

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

func 1（简单行向扫描转置）：正确性：1 func 2（列向扫描转置）：正确性：
1 func 3（使用之字形访问模式）：正确性：1

步骤 2：为已注册的转置函数生成内存轨迹。

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

正式提交的摘要（func 0）：正确率=1 未命中率=287

在本例中，我们在 trans.c 中注册了四个不同的转置函数。test-trans 程序测试每个注册函数，显示每个函数的结果，并提取结果用于正式提交。

下面是一些有关 B 部分工作的提示和建议。

- test-trans 程序会将函数 *i* 的轨迹保存在 trace.*fi* 文件中[2] 这些轨迹文件是非常宝贵的调试工具，可以帮助你准确了解每个转置函数的命中和失误来自何处。要调试某个特定函数，只需在参考模拟器上运行其跟踪文件，并使用verbose 选项即可：

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

- 由于您的转置函数是在直接映射高速缓存上求值的，因此冲突丢失是一个潜在的问题。请考虑代码中冲突未命中的可能性，尤其是沿对角线的冲突未命中。想出可以减少冲突丢失次数的访问模式。

- 阻塞是减少缓存缺失的有效技术。请参见

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

  了解更多信息。

[2]由于 `valgrind` 引入了许多与代码无关的堆栈访问，因此我们从跟踪中过滤掉了所有堆栈访问。这就是我们禁止使用局部数组并限制局部变量数量的原因。

6.3 把一切结合起来

我们为您提供了一个名为 `./driver.py` 的*驱动程序*，可对您的模拟器和转置代码进行全面评估。这也是你的指导老师用来评估你的模拟器的程序。驱动程序使用 `test-csim` 评估你的模拟器，并使用 `test-trans` 在三种矩阵大小上评估你提交的转置函数。然后，它将打印出结果摘要和你获得的分数。

要运行驱动程序，请键入

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

7 交作业

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

学校专用：在此插入文字，告诉每个学生如何在学校他们的用户名`--handin.tar` **文件。**

重要提示：不要在 Windows 或 Mac 机器上创建移交的压缩包，也不要移交任何其他压缩格式的文件，如 `.zip`、`.gzip` 或 `.tgz` 文件。