华中科技大学

计算机系统结构实验报告

院　　系　计算机科学与技术学院

专业班级　本硕博 2001 班

姓　　名　李茗畦

学　　号　U202015630

指导教师　万胜刚

2024 年 7 月 24 日

目 录

1 Cache 模拟器设计 2
1.1 实验目的 2
1.2 实验环境 2
1.3 实验思路 2
1.3.1 cache_line 结构的声明 2
1.3.2 cache 的初始化 3
1.3.3 cache 的仿真模拟 3
1.4 实验结果和分析 4
2 优化矩阵转置 5
2.1 实验目的 5
2.2 优化 32 × 32 矩阵 5
2.3 优化 64 × 64 矩阵 7
2.4 优化 61 × 67 矩阵 10
3 总结和建议 11

1 Cache 模拟器设计

1.1 实验目的

实现一个 cache 模拟器，实现一个简单的缓存。

1. 该 cache 需要支持读写操作并基于 LRU 策略处理缓存不命中的情况；
2. cache 模拟器从 trace 文件读取一系列读写请求，正确统计该缓存的命中，不命中以及驱逐次数，实现对计对计算机缓存的正确模拟；
3. 该缓存需要支持不同的 cache 参数，比如 cache 的相联度，cache 行的数目，一个 block 的大小，这些参数在调用这个缓存模拟器时被指定；

1.2 实验环境

1. 操作系统: Ubuntu 20.04 LTS
2. 处理器: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
3. 编译器: gcc (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0

1.3 实验思路

1.3.1 cache_line 结构的声明

一个 cache_line 表示我们的 cache 模拟器中存储的一个 cache 块。本实验只是对 cache 的模拟，因此它不需要存储一个数据块的真实数据。它应该包含三部分:

1. 有效位, 该成员变量用于表示该 cache_line 中存储的 cache 块是不是有效的
2. 时间戳, 该成员变量用于在 LRU 置换策略下淘汰 cache 块。每经过一个读写操作后, 我们将 cache 中所有有效的 cache_line 的时间戳加 1。
3. tag, 该成员变量表示 cache 块地址中的 tag 字段, 即前 64-S-E 位,S 和 E 为命令行参数，分别表示 cache 的 set 数目和相联度。

1.3.2 cache 的初始化

我们从命令行参数中读取相关参数，并据此声明一定大小的 cache_line。

1. 我们利用 getopt 命令对参数进行解析。其中参数 h 表示输出命令的使用说明；v 表示跟踪输出每次读写请求的模拟结果；s 表示 set index 的位数，即共有 2^s 个 cache set；E 表示 cache 的相联度，即一个 cache set 中 cache_line 的数目；b 表示 block offset 的位数，即一个数据块的大小为 2^b；t 表示 trace 文件的路径。
2. 在读取 cache 的参数后，我们声明一个二维 cache_line 型数组来表示需要模拟的 cache，数组的大小为 $E\times 2^s$, 并将数组中的每个 cache_line 的有效位都初始化为 0，表示 cache 为空。

1.3.3 cache 的仿真模拟

接下来开始从 trace 文件中逐行读取读取、写入或者修改操作。操作类型由每行的第一个字符决定。

1. find_data 函数：该函数用于在 cache 中寻找一个地址所在的 cache 块。该函数的参数为访问或写入内存的地址，首先经过移位操作计算出该地址的 set index 和 tag 字段，接着从对应的 cache line 集合中寻找 tag 字段相符且有效位为 1，如果找到则返回这个 cache line 的列索引，表示命中；如果没有找到则返回-1，表示未命中。
2. find_available_block 函数：该函数用于当不命中时，在 cache set 中寻找一个 cache line，如果有空闲的 cache line 则返回该空闲 cache line，否则根据 LRU 策略寻找该 cache set 中时间戳最大的 cache line。
3. L 型操作和 S 型操作：在 cache 模拟器中，这两种操作的行为是一样的。首先通过移位操作计算出操作地址的 set index 和 tag 字段，并调用 find_data 函数在 cache 中寻找该块。如果命中，则更新时间戳和 hit 计数。如果未命中，则调用 find_available_block 函数寻找一个 cache line 调入该块，并更新响应的 miss 计数和 evict 计数。
4. M 型操作：相当于一次 L 型操作加一次 S 型操作。

5. 更新时间戳：在对 trace 文件中的一行操作执行完毕后，需要更新 cache line 的时间戳。对 cache_line 数组中所有有效位为 1 的 cache_line 的时间戳加一。

1.4 实验结果和分析

运行代码包里的测试文件，运行结果如1-1所示。可见实验完成的 cache 模拟器和标准 cache 模拟器效果相同。

```
                     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

TEST_CSIM_RESULTS=27
```

图 1-1 cache 模拟运行结果

2 优化矩阵转置

2.1 实验目的

本节实验通过矩阵分块的技术，减少程序对 cache 的不命中次数，进而优化矩阵转置的性能。本节实验使用的 cache 为 s=5,E=1,b=5，即 block offset 为 5 位，set index 为 5 位，一个 cache set 有一个 cache line，也就是直接相联。

2.2 优化 32×32 矩阵

由于 block offset 为 5 位，因此一个 cache 块可以存放 32 个字节即 8 个 int 型数据。对于 32×32 的矩阵来说，每行的每 8 个 int 型数据属于一个 cache 块，在调入 cache 时将被一起调入。矩阵和 cache set 的对应关系如2-1所示, 其中一个矩形代表 8 个连续的 int 型数据，标号表示他们映射到的 cache set。由该图像可见，将 32×32 的矩阵划分为 8×8 的矩阵后，同一个子矩阵内的 8 行数据属于不同的 cache set，当将这 8 行数据都调入 cache 后，访问一个 8×8 的子矩阵内部是不会发生 cache 冲突的。

实现矩阵转置的最简单的函数如代码1所示。在访问一个 8×8 的子矩阵时只会在访问初发生强制不命中。在同一列的 4 个 8×8 的子矩阵是被映射到同一个 cache set 中。因此在代码1中，第一层循环遍历 A 矩阵的一行共 32 个 int 型数据，第二层循环遍历 B 矩阵的一列，这时每访问一个新的 8×8 的矩阵都会导致 8 次 cache 冲突。同时，A 和 B 矩阵在相同位置的元素也被映射到同一个 cache set 中，这也会导致 cache 冲突。因此该矩阵转置函数会导致较多的 cache 不命中。

为了适应 cache 块的大小，使得被调入 cache 的一个块中的数据被充分利用，防止同一个数据被反复调入 cache，我们利用分块技术，将 32×32 大小的矩阵划分为 8×8 的矩阵，8×8 的矩阵可以被完全调入 cache 中，除了调入时的强制不命中，在遍历 8×8 的矩阵的过程中是不会发生 cache 不命中的。

对于对角线上的 8×8 矩阵的转置，需要特殊处理。将一个 8×8 的矩阵被映射到的 cache set 看作一个集合，32×32 的矩阵和 cache set 的对应关系如2-2所示，每个矩形表示一个 8×8 的矩阵，具有相同标号的矩形表示该矩阵对应的

cache set 0	cache set 1	cache set 2	cache set 3
cache set 4	cache set 5	cache set 6	cache set 7
cache set 8	cache set 9	cache set 10	cache set 11
cache set 12	cache set 13	cache set 14	cache set 15
cache set 16	cache set 17	cache set 18	cache set 19
cache set 20	cache set 21	cache set 22	cache set 23
cache set 24	cache set 25	cache set 26	cache set 27
cache set 28	cache set 29	cache set 30	cache set 31
cache set 0	cache set 1	cache set 2	cache set 3
cache set 4	cache set 5	cache set 6	cache set 7
cache set 8	cache set 9	cache set 10	cache set 11
cache set 12	cache set 13	cache set 14	cache set 15
cache set 16	cache set 17	cache set 18	cache set 19
cache set 20	cache set 21	cache set 22	cache set 23
cache set 24	cache set 25	cache set 26	cache set 27
cache set 28	cache set 29	cache set 30	cache set 31
cache set 0	cache set 1	cache set 2	cache set 3
cache set 4	cache set 5	cache set 6	cache set 7
cache set 8	cache set 9	cache set 10	cache set 11
cache set 12	cache set 13	cache set 14	cache set 15
cache set 16	cache set 17	cache set 18	cache set 19
cache set 20	cache set 21	cache set 22	cache set 23
cache set 24	cache set 25	cache set 26	cache set 27
cache set 28	cache set 29	cache set 30	cache set 31
cache set 0	cache set 1	cache set 2	cache set 3
cache set 4	cache set 5	cache set 6	cache set 7
cache set 8	cache set 9	cache set 10	cache set 11
cache set 12	cache set 13	cache set 14	cache set 15
cache set 16	cache set 17	cache set 18	cache set 19
cache set 20	cache set 21	cache set 22	cache set 23
cache set 24	cache set 25	cache set 26	cache set 27
cache set 28	cache set 29	cache set 30	cache set 31

图 2-1　32×32 矩阵和 cache set 的对应情况

cache set 为同一个集合。当转置的 AB 中的矩阵没有位于对角线时，A 中子矩阵和 B 中的子矩阵总是具有不同的标号，因此 A 的子矩阵和 B 的子矩阵之间并不会 cache 冲突。当访问 A 和 B 的矩阵中位于对角线上的 8×8 的矩阵时，每一行的数据属于同一个 cache set，且本实验的 cache 为直接相联，在处理对角线上的矩阵时，当我们读取 A[i][i] 后，将该值写入到 B[i][i] 中时，会导致 A 矩阵中这一行的 8 个数据所在的 cache 块被 cache 驱逐，这会导致较多的冲突不命中。因此，对于 A 中处于对角线上的 8×8 矩阵中的每行元素，我们将这一行的元素一次全部读出，放置在 8 个局部变量组成的缓冲区中，这样在读取矩阵 B 中的元素时，即使发生冲突导致 A 中的数据被驱逐，他们将来也不会被访问到。每 8×8 的子矩阵的转置过程中读取 A 矩阵会发生 8 次强制不命中，向 B 矩阵写入会发生 8 次强制不命中，32×32 的矩阵转置过程中一共会发生 256 次 cache 不命中。最终 32×32 的矩阵的转置过程中 cache 的运行结果如2-3所示，发生的 cache miss 次数为 259，其中 256 次是矩阵转置造成的，3 次为函数调用造成的额外开销。

Listing 1　简易的矩阵转置函数

```
1  void trans(int M, int N, int A[N][M], int B[M][N]){
2      int i,j,tmp;
3      for(i=0;i<N;i++){
4          for(j=0;j<M;j++){
5              tmp=A[i][j];
6              B[j][i]=tmp;
7          }
8      }
9  }
```

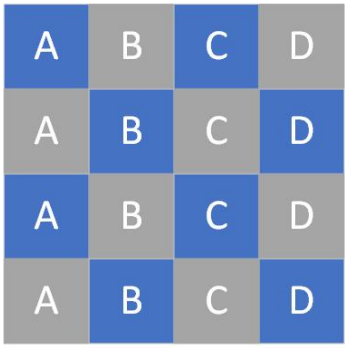

图 2-2　32×32 和 cache set 的对应情况

2.3　优化 64×64 矩阵

在 64×64 矩阵中，我们采用的 cache 仍然是块大小为 32 个字节的，因此我们仍希望一次完成同一行的 8 个数据的转置。当我们对 8×8 的 B 矩阵按列访问时，这个矩阵中 8 行数据与 cache set 之间的对应关系如2-4(以第一个 8×8 矩阵为例) 所示。前 4 个数据和后 4 个数据分别被映射到同一个 cache set 中，在访问一列的 8 个数据的过程中就会产生 4 次强制不命中和 4 次冲突不命中，在访问

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:2242, misses:259, evictions:227
```

图 2-3　32×32 矩阵转置结果

第二列时又会产生 8 次冲突不命中。这样，B 矩阵中的每个数据写入时都会导致一次不命中。

为了防止在一个 8×8 内部中发生 cache 冲突，我们在其内部再次划分为 4×4 的矩阵。不过，cache 中一个块为 8 个 int 型数据，当我们完成了一个 4×4 的矩阵的转置后，在这个矩阵之后的 4×4 的矩阵也被顺带放入了 cache 中，如果我们此时直接对下一个 4×4 的矩阵进行转置，那么 A 和 B 之中总有一个要将被顺带进入 cache 的 4×4 的矩阵从 cache 中驱逐，之后用到这个矩阵时需要重新调入造成多余的 cache 不命中。因此，我们应尽可能的利用已经进入 cache 的块的空间。

将 8×8 的矩阵划为 4 个 4×4 的矩阵，分别记为 A[1,1],A[1,2],A[2,1],A[2,2], 表示左上，右上，左下和右下的小矩阵。当完成 A[1,1] 到 B[1,1] 的转置后，A[1,2] 和 B[1,2] 也已经进入了 cache，如果此时直接进行 A[1,2] 到 B[2,1] 的转置，B[1,2] 会被驱逐出 cache，为了利用 B[1,2] 的空间，我们先将 A[1,2] 转置放置到 B[1,2] 中。此时 A[1,1] 和 A[1,2] 已经访问完毕，便可以被 cache 驱逐了。之后我们进行 A[2,1] 到 B[1,2] 以及 B[1,2] 到 B[2,1] 的转置。首先将 B[1,2] 的一行读入到一个大小为 4 的 buffer 中，接着从 A[2,1] 中读取一列放入 B[1,2] 对应的行中，然后把 buffer 中的数据放入到 B[2,1] 对应的行中。B[1,2] 和 B[2,1] 中的同一行映射到同一 cache set，B[1,2] 中的一行的数据通过 buffer 拷贝，因此这里每一行只会造成一次不命中。对 B[1,2] 和 B[2,1] 中的每一行拷贝完成后，再进行 A[2,2] 到 B[2,2] 的转置，此时这两个块都位于 cache 中，不会造成不命中。8×8 的矩阵的转置过程如2-5所示。

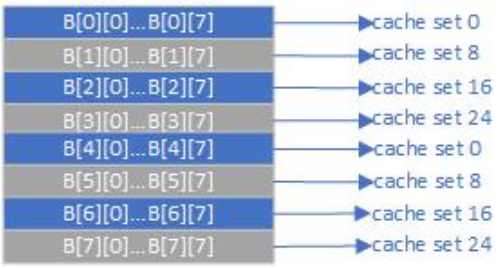

图 2-4　64×64 的矩阵中一个 8×8 矩阵和 cache set 的对应情况

和 32×32 的矩阵类似，A 和 B 矩阵在处理非对角线上的块时彼此之间仍然不会发生 cache 冲突。但是对于处于对角线上的块时，不仅在 8×8 矩阵内部会

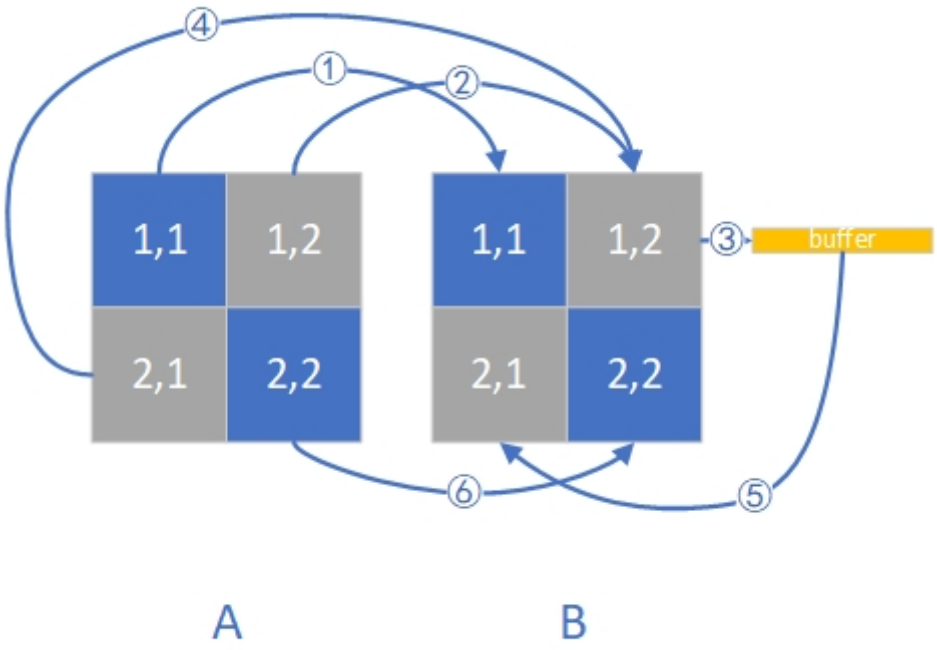

图 2-5　8 × 8 矩阵转置示意图

产生 cache 冲突，A 和 B 矩阵之间也会产生 cache 冲突。2-5中的过程中 1 和 2 之间，4 和 5 之间以及 6 的过程中会产生大量的冲突不命中。

这种情况下不能再把 B 中的空间作为矩阵 A 的缓存区，为了避免对角线的情况下的冲突不命中，我们把 B 矩阵中下一个将要进行转置的 8 × 8 矩阵作为缓存区[1] 。此时转置过程的示意图如所示。C[1,1] 和 C[1,2] 表示的是 B 矩阵中下一轮循环将要用到的 8 × 8 矩阵的左上角和右上角的块。为了避免 A 矩阵中 A[1,1],A[1,2] 和 A[2,1] 和 A[2,2] 之间的冲突，我们首先把 A[2,1] 和 A[2,2] 拷贝到 C[1,1] 和 C[1,2]。之后再将 A[1,1] 和 A[1,2] 中的内容逐行拷贝到 B[1,1] 和 B[1,2]，此时 A[2,1] 和 A[2,2] 会被 cache 驱逐，拷贝完成后，A[1,1] 和 A[1,2] 也会被 cache 驱逐。

之后将 B[1,2] 中的数据拷贝到 B[2,1] 中，这个过程利用 buffer 逐行拷贝，每将 B[1,2] 中的一行读出到 buffer 中后，立即将 C[1,1] 中的一行拷贝回 B[1,2]，之后将 buffer 中的数据拷贝到 B[2,1] 中，此时 B[1,2] 中该行会被 cache 驱逐。最后，将 C[2,2] 中的数据拷贝到 B[2,2] 中。

对角线上的最后一个 8 × 8 矩阵没有其他的 8 × 8 矩阵作为其缓冲区，因此

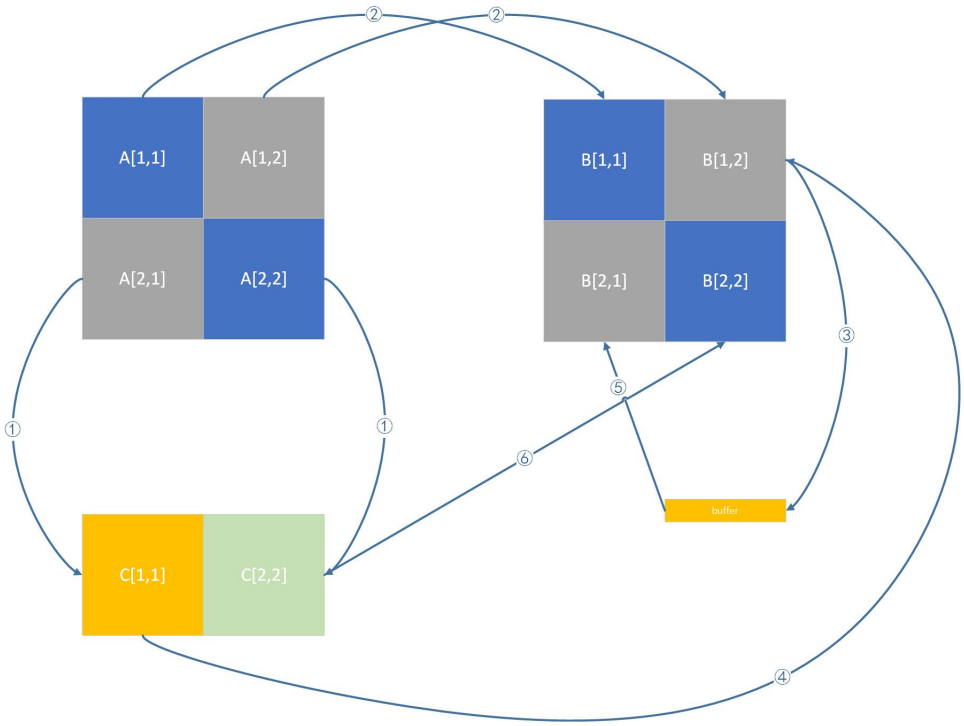

图 2-6　对角线 8×8 矩阵转置示意图

它只能采取2-5表示的过程。64×64 矩阵转置的运行结果如所示，会产生 1046 次 cache 不命中。11+27+11。对于非对角线上和对角线上经过优化的 8×8 的矩阵转置时会产生 16 次 cache miss，那么 64×64 会产生 1024 次 cache miss，由于对角线上最后一个 8×8 的矩阵以及函数调用的开销，会增加 22 次 cache miss。

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:10403, misses:1046, evictions:1014
```

图 2-7　64×64 矩阵转置运行结果

2.4　优化 61×67 矩阵

对于 61×67 的矩阵转置，我们依旧采用矩阵分块的技术。通过尝试分块的大小，将分块设为 14 可以将不命中次数降为 2000 次以下。61×67 矩阵转置运行结果如2-8所示。

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6183, misses:1996, evictions:1964
```

图 2-8　61 × 67 矩阵转置运行结果

3　总结和建议

经过本实验，自己对 cache 是如何实现的有了更深刻的理解，学习了如何利用 cache，通过分块技术减少程序的不命中次数从而优化程序性能。

希望本课程对其他章节也设置一些实验，如果只上理论课的话感觉对技术的理解始终比较肤浅。

参考文献

[1] http://csapp.cs.cmu.edu/3e/cachelab.pdf

[2] https://zhuanlan.zhihu.com/p/79058089

[3] http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/recitations/rec07.pdf