华中科技大学

计算机系统结构实验报告

姓　　名：　　徐雨梦

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2009

学　　号：　　U202015562

指导教师：　　王芳/陈俭喜

分数	
教师签名	

2023 年 4 月 15 日

目　录

1. Cache 模拟器实验

1.1. 实验目的

1. 理解 cache 工作原理；
2. 实现一个高效的 cache 模拟器。

1.2. 实验环境

Linux 64-bit，C 语言。

1.3. 实验思路

1. 数据结构设计与整体流程

（1）cache 工作原理：

该实验中采用了组相联映射 cache：cache 被分为多个组，每个组中有几行。同时访问的数据的地址分为 3 个部分，即为(标记位，组号，块内地址)，其中块内地址在查询 cache 的时候可以忽略，重要的是组号与标记位，其中组号可以直接索引到 cache 的一部分数据块，接下来组内保存的几行数据同时与标记位进行比较，找到相等且有效的数据即为命中。

（2）LRU 替换策略

这里对于冲突缺失采用 LRU 策略进行数据块替换，也就是将最久不使用的块驱逐，由于是模拟 cache，可以在查找数据的时候更新最久未使用次数。

（3）数据结构

根据组相联 cache 的原理和 LRU 原理可以知道，cache 中的一个块应该包含的信息有：①有效位；②计数器；③访问的位置。此处的访问位置不是真实的访问位置，因为需要忽略块内地址，所以需要保存偏移后的地址，可以是(标记位)，也可以是(标志位，组号)。

```
typedef struct cache{
  int valid;//有效位
  int count;//计数多久没用了
  unsigned long long addr;//缓存的位置,偏移之后的
}CACHEBLOCK;
```

（4）整体流程

在整个过程中，需要分析执行函数时传递的参数并打开文件进行 cache 模拟，因此对于实验的整体流程为：

①接收命令行参数并解析，初始化一些变量；

②为模拟的 cache 分配空间；

③打开文件，以此读取指令对 cache 读写进行模拟并计数命中、缺失、驱逐的次数。

2. 命令行解析

实验在 Linux 环境下运行，在执行函数的时候采用命令行实现，因此需要对 main 函数修改为带输入参数的函数。之后就可以调用 getopt 函数循环读取命令行中的标志和对应参数，读取的过程中，需要保存偏移宽度（块内地址占有的位数）、cache 的组数与每组的行数、要打开的文件夹名称。

3. 设置 cache

在命令行解析的时候，可以得到组数和每组的行数，因此可以对其采用 malloc 函数分配一个二维数组进行 cache 模拟。而同时也可以采用一维数组进行模拟，我们知道二维数组的内存空间与相同大小的一维数组相同，在访问一组 cache 的时候，都是访问连续的一块地址，因此也可以采用一维数组实现，但需要注意访问的数组的开始位置为组号*行数。

设置好了之后将所有的 cache 内的数据初始化为 0。

4. 依次执行指令进行模拟

采用 fopen 并用只读模式“r”打开文件，采用 fscanf 循环读取文件中的指令，该实验中忽略了其他的参数，只考虑 L、M、S 三种命令。查找 cache 并更新的操作流程为：

①获取访问的地址的组号，即右移忽略块内地址，并根据参数只取地址的低 s 位；

②获取标志位，即右移掉块内地址和组号；

③根据组号，遍历该组的块：如果有效位为 0，记录下该空闲块；如果有效并且数据与标志位相同，设置命中标记为 1，并将该块的计数器清零，同时对命中计数加一；如果有效但数据与标志位不同，将该计数器加一，并比较该计数器与最大计数器，保存最大的计数器的位置。

④遍历完块后，如果没命中，缺失计数加一，同时查看是否记录下来了空闲块，如果有就将标志位保存在该块中，没有就将数据保存在记录的计数器最大值的位置中；否则继续。

1.4. 实验结果和分析

在 Linux 环境下 make 并运行 test-csim，模拟结果与对照结果一致。

```
daydream@DESKTOP-NOQDMH7:~$ cd cachelab-handout/
daydream@DESKTOP-NOQDMH7:~/cachelab-handout$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     2 (1,1,1)       8       8       6       9       8       6  traces/yi2.trace
     2 (4,2,4)       2       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     2 (2,1,3)     147      71      67     167      71      67  traces/trans.trace
     2 (2,2,3)     181      37      29     201      37      29  traces/trans.trace
     2 (2,4,3)     192      26      10     212      26      10  traces/trans.trace
     2 (5,1,5)     211       7       0     231       7       0  traces/trans.trace
     4 (5,1,5)  246213   21775   21743  265189   21775   21743  traces/long.trace
    19

TEST CSIM RESULTS=19
```

图 1 csim 测试结果

2. 优化矩阵转置

2.1. 实验目的

1. 理解 cache 工作原理；
2. 实现对矩阵的分块转置，降低访问矩阵时的 cache 缺失命中数。

2.2. 实验环境

Linux 64-bit，C 语言。测试调用了实验一中的模拟进行了计数，并且测试的参数为(5,1,5)。

2.3. 实验思路

1. 缺失原因分析与优化方向

实验在测试的时候设置参数为(5,1,5)，因此 cache 有 32 组，每组只有一个块，每个块大小为 32 字节，由于这里的矩阵是 int 型，因此每个块可以保存矩阵中的 8 个数据，因此只要有两个数据映射到同一地址，就会发生冲突缺失。

如果只是简单的实现矩阵转置，会因为总是访问到相同映射位置的块而不断的产生冲突缺失，对此的解决方案就是针对每次的访问，将访问到的块的数据保存下来，这些数据应该会保存在寄存器中或者另外的 cache 块中从而避免了相同的映射位置。

对于矩阵转置来讲，总是会有一个数组并不能被连续的访问而是跳跃的访问，因此对几个数赋值的过程中，就会产生多次缺失，而由于冲突缺失，这些块在被载入 cache 之后和下一次被访问之前容易被驱逐，从而造成下次无法命中。

优化的方向就是对矩阵分块，分块转置。而分块的大小取决于 cache 的大小。这里 cache 每个块可以保存 8 个数据，因此如果可以的话，分块至少为 8*8，不然一次访问的数据并没有使用就被替换掉了，降低了命中率。

2. 具体实现：

由于交替 cache 的一个组的解决方案为保存数据，这个只需要一个大小为 8 的数组即可实现。所以只详细介绍分块的优化方向的具体实现。

（1）32*32 矩阵：

分块方案：分成 8*8 的小矩阵，对每个小矩阵进行转置。

具体原因：cache 的每个块都可以承载 8 个数据，并且有 32 组，那么对于一个矩阵，刚好可以承载 8*32 的数据。那么在 A 数组访问的时候，每缺失一次载入 8 个数据，这 8 个数据刚好是 B 数组试图写入的一竖行的 8 个上下连续的数据，而这 8 个数据所在的块所映射的 cache 的组恰好并不相同，因此分块为 8*8 的情况下，B 的块要么一直在 cache 中，要么被驱逐一次再重新载入后一直保存着。刚好可以在读取每一组数组的时候将其写入 B 在 cache 中对应的块中。

（2）64*64 矩阵：

分块方案：分成 8*8 的小矩阵，对每个小矩阵进行转置；在对 8*8 矩阵进行转置的过程中，将其再次分为 4*4 的小矩阵进行局部转置和块数据交换。

具体原因：

64 的矩阵一行可以分成 8*8，因此 cache 对应于 4*64 的数据块。在 A 数组访问缺失载入 8 个数据的时候，B 数组试图写入的一竖行的 8 个数据所在的块所映射的 cache 的组互相重复，因此如果将其按照 8*8 对一竖行依次访问将会产生大量的冲突缺失，而竖行的 4 个对应的 cache 块并不冲突，因此可以将其分成 4*4 的四个块进行转置。

这四个块从左上开始标志 1、2、3、4，那么需要做的就是将四个块转置并进行赋值：A1->B1,A2->B3,A3->B2,A4->B4。而由于每次访问数据都是按照块大小访问的，所以 A 每次获取都会涉及横着的两个块的数据，因此流程为：

①将上部的两个块转置：A1->B1,A2->B2；

②接下来对 B2->B3,A3->B2；

③A4->B4。

（3）61*67 矩阵：

分块方案：分成 16*16 的小矩阵，对每个小矩阵进行转置；对于剩下不能分成 16*16 的划分为右部和下部两部分进行转置。

具体原因：

在每次 cache 载入数据时，仍旧按照 8 个数据为一块载入，但由于宽度为 67，因此 cache 在载入数据的时候并不能跟数组的宽度对齐载入。此时如果按照 8*8 进行分块的话，很多时候载入的 8 个数会处于两个块中，载入的第 2 个块的数据中有一部分在下一个 8*8 的块中，但是由于对 A 和 B 两个数组的 8*8 访问，原本 A 中访问的第二块可能会被 B 和少量的 A 中的块驱逐，导致在对下一个 8*8

的块进行访问的时候再次缺失，因此可以一次访问多个块来减少冲突访问。此时考虑 B 数组竖着一次性访问 8/16/..的数据，发现由于宽度未与 8 对齐使得每一竖行对应的映射位置冲突的概率很小，因此访问多个 8 不会对 B 造成多大的冲突缺失影响。

但是这里一次载入两个块，因此小于 61 和 67 的可以以 8 的多倍数为公因数的最大数值为 48 和 64，最大公因数为 16。在访问完 64*48 后，剩余 15*67 与 3*48 两个大块，对于 15*67，每一行都涉及两个不同的块，直接一行为单位进行置换即可；对于 3*48，一块一块地按照 8 个数据进行置换即可。

2.4. 实验结果和分析

实验结果：

32*32：缺失次数 290<300;

64*64：缺失次数 1182<1300；

61*67：缺失次数 1926<2000。

```
daydream@DESKTOP-NOQDMH7:~/cachelab-handout$ ./test-trans -M 32 -N 32

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:1765, misses:290, evictions:258

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:869, misses:1184, evictions:1152

Summary for official submission (func 0): correctness=1 misses=290

TEST_TRANS_RESULTS=1:290
```

图 2 32*32 矩阵

```
daydream@DESKTOP-NOQDMH7:~/cachelab-handout$ ./test-trans -M 64 -N 64

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:9065, misses:1182, evictions:1150

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3473, misses:4724, evictions:4692

Summary for official submission (func 0): correctness=1 misses=1182

TEST_TRANS_RESULTS=1:1182
```

图 3 64*64 矩阵

```
daydream@DESKTOP-NOQDMH7:~/cachelab-handout$ ./test-trans -M 61 -N 67

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:6255, misses:1926, evictions:1894

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3755, misses:4424, evictions:4392

Summary for official submission (func 0): correctness=1 misses=1926

TEST_TRANS_RESULTS=1:1926
```

图 4 61*67 矩阵

3. 总结和体会

这次的实验实现了一个模拟的 cache 并模拟 cache 的访问过程和一个优化的矩阵转置，并且第二个实验基于第一个实验来测试性能。两个实验都围绕着内存访问和 cache 的原理。

第一个实验比较简单，只需要模拟即可，重要的是解析参数，需要对相关函数比如 getopt 等熟悉。第二个实验较有难度，优化有多种方案，如何最大化的利用 cache 需要非常熟悉 cache 的原理和数组的数据组织方式并且有技巧的划分块进行转置操作。

在本实验中，我查询了网上较为高效的优化方案并实现了一种，最后达到了该实验的满分标准。在这个过程中对 cache 的原理和数组的组织方式和访问有了更深的认识，同时认识到了平时非常常见的数组访问也可以进行优化进一步的加速计算机的运行速度，对计算机的一些细节问题认识到了重要性。

4. 对实验课程的建议

对于该实验，实验说明的课件阐述的不够清晰，有一些点比较迷惑，比如替换策略并没有说明是 FIFO 还是 LRU 还是其他，在助教发了原版实验的课件才更加清晰一些。我觉得可以把实验说明写的更清晰一些，这样学生的实验效率也会更高。