Lab4: 缓存模拟器

匡亚明学院　181840326　张祎扬

思考：数据对齐和存储层次结构

数据对齐有利于提高CPU访问数据的效率。而数据不对齐的时候在读取数据时可能要进行多次访存，从而增加开销。比如假如每次读地址都从偶地址开始，但是一个int型变量存储在奇地址，就需要读两次并且把两次读到的数据拼接成一个int型数据，这就增加了访存开销。同时数据对齐还有利于节省存储空间，最大化利用空间。

实现cache

首先声明了一些结构，用来表示cache的组，行等，又在行里设置了valid位，dirty位等。又在cache中记录了组的数量和每组的行数。

init_cache

两个参数分别表示表示cache大小的数据的总宽度和关联度的数据的宽度。用malloc申请空间，并且将有效位和tag全部置0.

cache_read

根据组相联映射的规则，在读的过程中，先判断对应的组中有没有相应的tag和有效位（在实际实现的时候用一个for循环进行遍历），如果有的话就直接命中，并且根据块内地址取出相应的信息。如果没有命中的话分两种情况。如果当前组内有空行，则直接读出主存块到该空行；如果当前组内没有空行，就随机选一组进行替换。

在实际实现的时候遇到了一些问题，因为要一次性读出四字节的数据，而data的数据类型我设置的是uint8_t的数组，所以需要读出连续读出数组的四个元素，并且进行拼接。而写的时候也是一次性写四个元素。所以我先写了两个函数用来封装读和写的功能，在调用它之前我只需要提供组号和行号和地址即可。

cache_write

还是通过循环遍历来判断有没有命中，如果命中就写进cache。若写不命中，先在cache中分配一行，将主存块调入该行中，并更新cache中相应单元的内容。如果没有空闲行的话就比较麻烦，不仅需要随机选一行替换，并且还要根据修改位判断有没有修改过，如果修改过则需要把它先写回主存。

具体的代码见.c文件。

我按照了自己的思路完成了全部代码后，还是不断触发assert。我打印了一些中间值出来看，找了很久的bug。最后给我启发的是最上面的那道思考题，因为读的时候每次读四位，然后我发现我并没有注意到数据对齐的原因，要把地址转化为4的整数倍。（其实也是从mem_uncache_read()函数得到的启发）

还有一个bug是在写读函数的时候，如果要进行随机替换，并没有判断dirty位并把cache写回内存。

改完bug以后，进行了几次随机测试，均通过测试。

```
katherine@debian:~/ics-workbench/cachesim$ ./cachesim-64
random seed = 1577007738
Random test pass!
katherine@debian:~/ics-workbench/cachesim$ ./cachesim-64
random seed = 1577007740
Random test pass!
katherine@debian:~/ics-workbench/cachesim$ ./cachesim-64
random seed = 1577007742
Random test pass!
katherine@debian:~/ics-workbench/cachesim$ ./cachesim-64
random seed = 1577007744
Random test pass!
```

观察cache的性能指标

为了计算命中率，我需要记录总的次数和命中的次数（或者没有命中的次数）。框架代码中提供了一个静态变量`cycle_cnt`，还提供了一个设置它的函数`cycle_increase()`，我发现在cpu_read和cpu_write中函数的参数都是1，而mem_read和mem_write中的参数要大得多，所以我觉得这个参数应该代表读写时间，在内存中的读写确实比cpu中开销大得多。最后我将这些指标在display函数中输出到终端。

输出一下看看，发现随机种子的命中率在83%左右。

```
katherine@debian:~/ics-workbench/cachesim$ ./cachesim-64
random seed = 1577009429
Random test pass!
the cycle_cnt is 56624111       the total count is 9388608      the missing count is 1523825
RATE: 0.837694
```

建模估计性能

根据讲义要求，要对以下几个方面进行建模：

1. cache的复杂性，具体度量为访问cache的时间。

   因为采取的是组相联映射，在访问的时候是对组内的各行进行遍历，所以组内的行数越多，访问cache的时间就越长，所以我在for循环的每一步循环都记一次cycle_cnt。

2. cache缺失时的代价，根据内存访问的时间建模。

   因为每一次访问内存都要读或写`BLOCK_SIZE`大小的数据，所以访问时间应该与它成正比，在这里我就直接乘以`BLOCK_WIDTH`了。

接下来就是修改参数改进模型了。根据我对cache的理解，对cache的性能有影响的因素有：块的大小（即一次性读入的数据多少），cache的总大小（最多可以缓存多少数据），关联度（每一组中的行数），以下分别进行探究。

当前的默认配置是块大小64B，cache总大小16KB，4路组相联。

首先保持默认总大小，关联度，改变块的大小。

BLOCK_WIDTH	命中率	平均时间
4	0.938634	11.344708
5	0.969246	8.371602
6	0.984532	6.391729
7	0.992203	5.177239
8	0.996030	4.438704
9	0.997951	3.990358
10	0.998899	3.798818

可以看出，随着块大小的增加，命中率不断提高，总体的代价也在降低。这是因为虽然块的大小增加了，访问主存的代价也提高了，但是因为命中率提高，访问主存的次数减少了，所以总的代价还是在减少的，平均时间也逐渐下降。不过在BLOCK_WIDTH达到8以后，无论是命中率还是性能的提升都有所减缓。

接着保持块宽度为6，四路组相联，改变cache的总大小。

cache总大小	命中率	平均时间
1KB	0.982971	6.761193
2KB	0.983763	6.623635
4KB	0.984166	6.534465
8KB	0.984404	6.443746
16KB	0.984547	6.405598
32KB	0.984738	6.336435
64KB	0.984850	6.250164

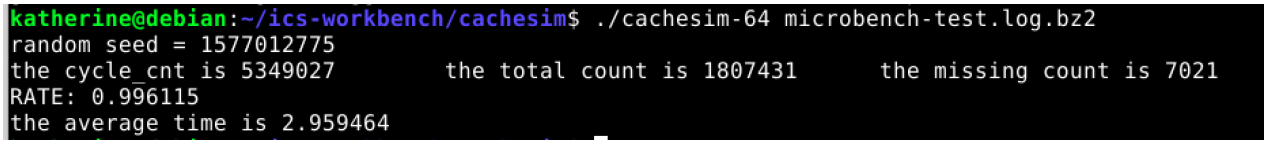
可以看到的是，随着cache总大小的增大，命中率不断提升，平均时间也在减少。但二者的提升都是非常小幅度的，尤其是cache超过32KB以后命中率的提升非常平缓。

接着保持宽度为6，cache总大小为16KB，改变关联度。

关联度	命中率	平均时间
1	0.984484	4.895407
2	0.984518	5.395846
4	0.984530	6.405783
8	0.984541	8.551418
16	0.984533	12.689593

可以看出关联度大了开销也随之增大，命中率的提升基本不明显，并且当关联度过大时命中率开始下降。

综合以上各个结果，我认为块宽度取8，cache总大小取64KB，关联度取1（直接映射，因为关联度对开销的影响比较大）比较合适。采取以上配置以后，命中率提升到了0.996115，而平均开销减少到了2.959464cycle（按照我的建模方法计算得到的结果）。



实验心得

其实实现cache本身不难，关键是要理解清楚cache的运行机制，怎么从地址中获取块内地址、组号等等。以及命中和不命中时分别怎么进行处理，什么是写回和随机替换，什么时候应该写回主存而什么时候不用写回主存。

实验前期我花了大量时间理解cache的机制，我反复看了教材和实例，还看了一点mooc上的讲解。把整个运行机制想清楚以后再开始写，思路就会清晰很多。多亏了做PA时痛苦地实现了分页机制，对我现在实现cache很有帮助。

另外，输出调试法真的很有用。