Lab4:缓存模拟器

——181220010 丁豪

实现Cache

根据提示有三点要求，Cache块大小为64B，随机替换，回写法。我们重点讲一讲这三项在代码中的实现，其余实现详见代码注释。

对Cache的结构进行建模，Cache是由行构成的，每一行除了数据块外还有各自的tag,valid,dirty三个属性，分别表示装入内存的高位，有效为，修改脏位。数据块，根据cache_write函数原型的提示，操作的最小单位为uint32_t，于是我们用uint32_t数组data来表示数据，其长度为64/4=16。

```
struct Cache
{
  bool valid,dirty;  //  有效位，脏位
  uint32_t tag;      //  标记，代表取模后的内存地址高位
  uint32_t data[16]; //  代表块内地址
}line[total_cash_line];//代表行号
```

随机替换的实现，使用rand（）函数生成0-0x7fff的随机数，并模cache组内行数，得到该组将被替换掉的行。

```
  int line_out = group_start + rand()%GROUP_LINES;
  c2m(group_num,line_out); // c2m为  memory to cache
  m2c(addr,line_out);      // m2c为  cache to memory 其涉及回写法
```

回写法的实现，在cache回到主存时，根据dirty位是否为1，采取写回操作。其余时候，直接操作cache，不改变主存对应部分的值。

```
if(line[line_index].dirty){
    // 主存地址某一行起始等于，tag与组号的拼接
    uintptr_t start = (line[line_index].tag << group_width)+group_num;
    mem_write(start, (uint8_t*)&line[line_index].data[0]);
    line[line_index].dirty = false;
}
```

性能评估

按照要求有两项参数需要建模：

1. Cache的复杂性，即CPU访问Cache的时间

为了评估Cache与内存访问的性能差异，首先建立cache耗时与内存耗时的对照标准。仿照lab3性能评估的方法，我们使用gettimeofday函数来获取进行cache操作的总微秒数，并于最终计算cache平均访问以此所消耗的微秒数。

2. cache miss时的代价，根据内存访问时间建模

因为这里的内存读取和cache读取并没有硬件差异，时间应该是差不多的。于是我们人为设置一个模拟时间cycle_cnt，置Cache访问的cycle_cnt开销1，内存读取为6，内存写入为25。以此来统计使用Cache的相对开销大小，也可以对比cache hit 与 miss不同情况下的开销。在实际操作时，将Cache占用的时间除以总时间，得到Cache使用的时间的占比，作为评价指标。

将上述两项开销整合起来，得到估计时间workload。其计算表达式为

$$
\begin{aligned}
workload &= 一次cache操作的开销 * 访问Cache的占比 + 一次内存操作的开销 * 访问内存的占比 \\
&= 一次Cache操作的开销 * 访问Cache的占比 + （1 - 访问Cache占比）
\end{aligned}
$$

3. 此外还添加了Cache的命中率作为另一指标。

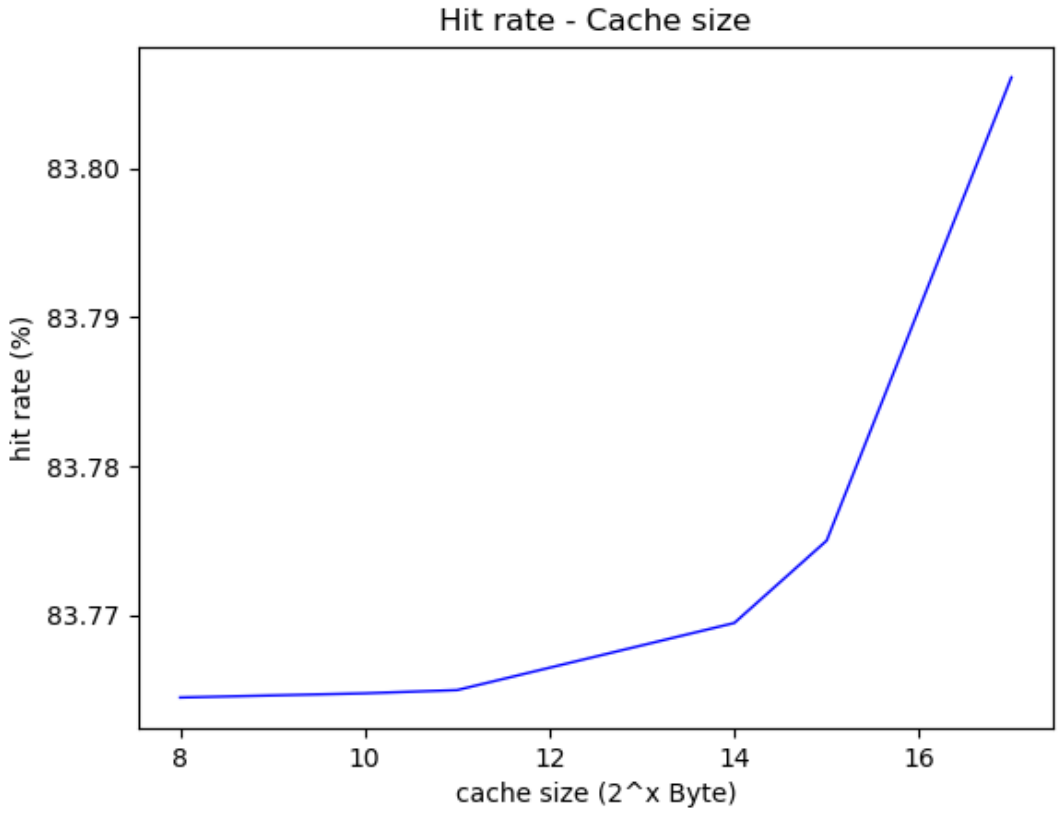
```
void display_statistic(void) {
  double mean_time = (double)real_time/(hit+miss);
  double cache_percent = (double)100*hit/cycle_cnt;
  double hit_rate = (double)100*hit/(hit+miss);
  printf("Cache mean time:%f\n",mean_time);
  printf("Cache time percent:%f\n",cache_percent);
  printf("hit_rate:%f\n",hit_rate);
  printf("Workload:%f\n",mean_time*cache_percent+(100-cache_percent));
}
```

为了保证每次测试的时间有可比性，使用同样的随机种子（-r 42）。

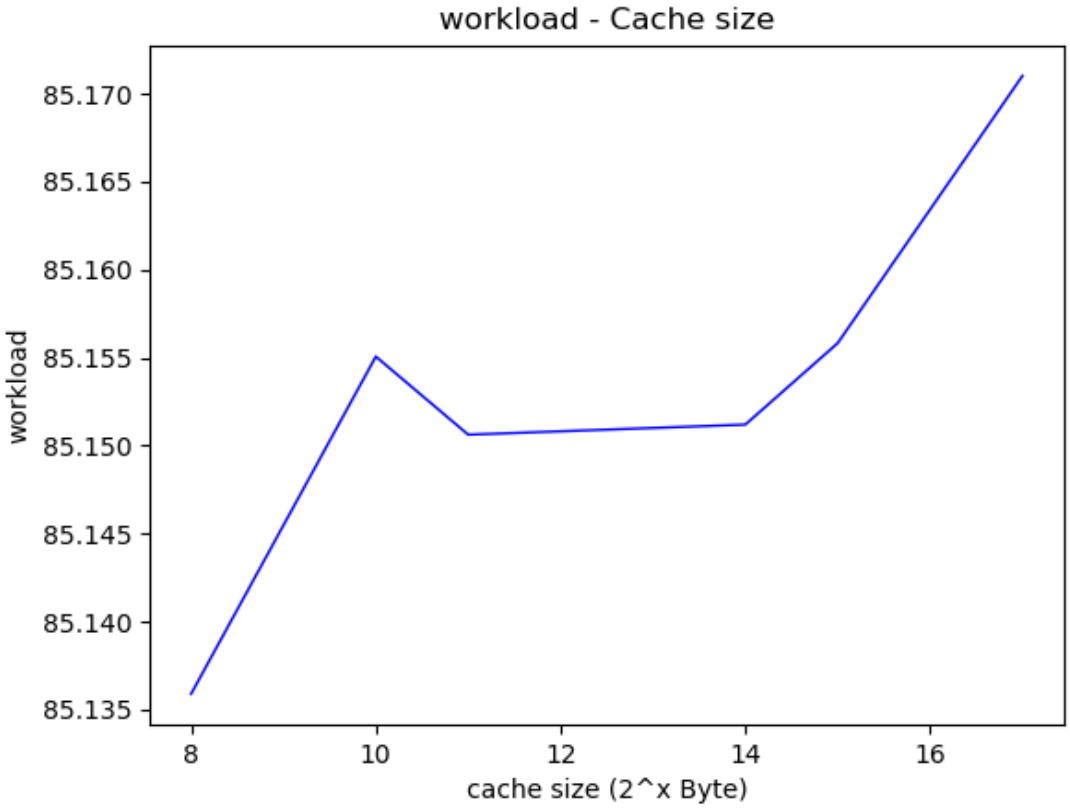
实验一、Cache总大小对Cache性能的影响

控制Cache的组相连为2路，改变Cache总大小。

Cache总大小(B)	Cache命中率(%)	workload
2^8	83.7645	85.135891
2^10	83.764782	85.155064
2^11	83.7650	85.150624
2^14	83.7695	85.151196
2^15	83.775028	85.155826
2^17	83.8061	85.171012



我们用python绘制图像，可以比较直观的看出Hit_rate与Cache_size的数量级大致呈现出指数级上升的正相关性。

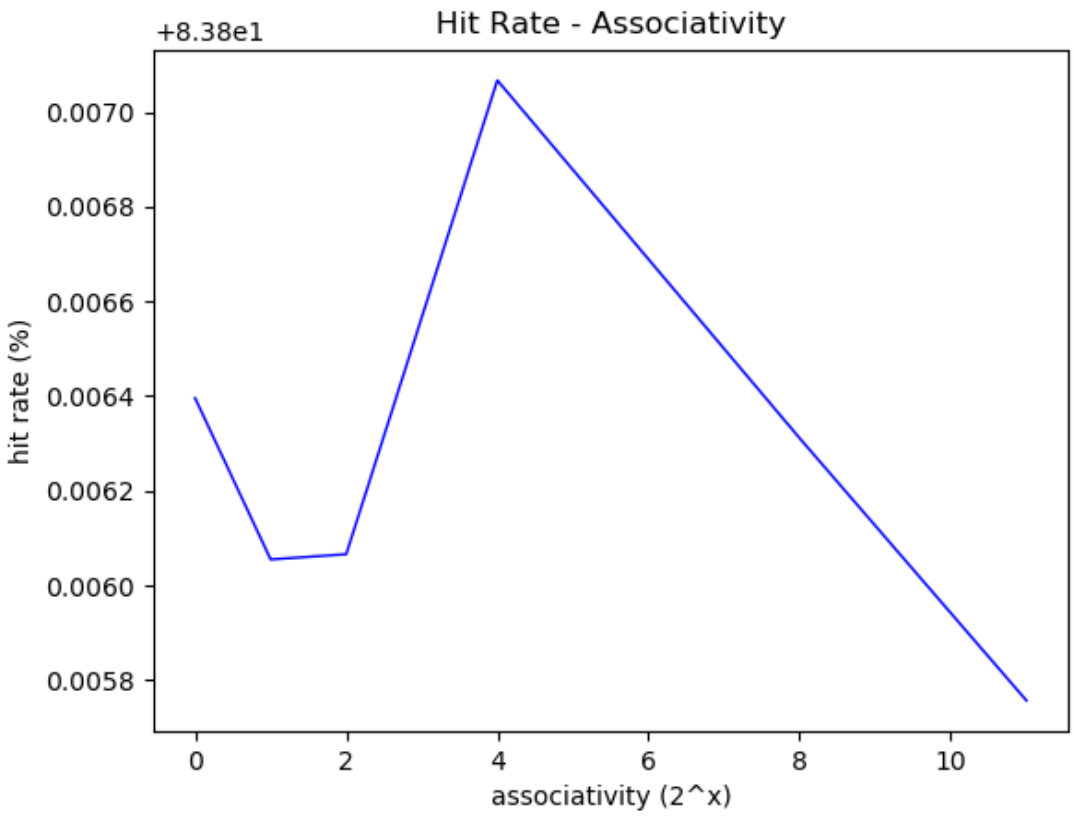


workload与cache_size大致呈正相关，但是通过实验发现，每次运行Cache的时间开销差异较大，可能与CPU运行状态等硬件状态有关，因此数据的准确性难以有稳定的保障。

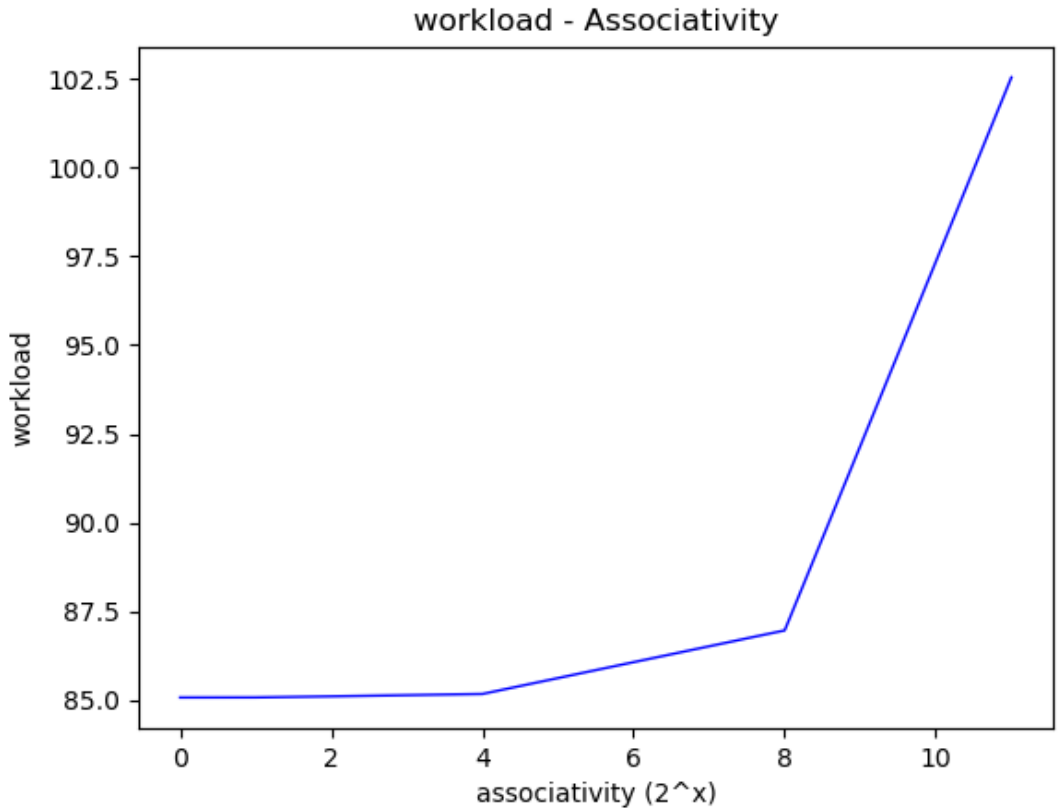
实验二、Cache组相连度对Cache性能的影响

控制Cache的大小为2^17，改变Cache组相连度。

Cache路数	Cache命中率(%)	workload
0	83.806396	85.074811
1	83.806055	85.077210
2	83.806066	85.107329
4	83.807067	85.174900
8	83.806311	86.964767
11	83.805757	102.537629



在改变组相联度的时候，发现命中率与之无明显相关性，4时取到最大值。



workload与上一个实验中的比较倾向于随机不同，在此表现出了较强的正相关性（足以抵消误差带来的影响），且可以发现在4与8两处出现了两次非常明显的斜率上升现象。

结论

增加Size可有效增加命中率但会导致时间开销有小幅上升，增加associativity对命中率的增加无明显关系，且会显著提升开销。综上考虑，**使用2^17Byte的Cache，4路组相联**，可以得到最佳效果。

思考题: 数据对齐和存储层次结构

想一想, 为什么编译器为变量分配存储空间的时候一般都会对齐? 访问一个没有对齐的存储空间会经历怎么样的过程?

进行对齐，可以使得程序的空间局部性更加优化，在载入进Cache时，不容易出现由于一个变量分属两个内存块而导致需要载入两块的尴尬情况。以此提高了Cache的命中率和访问效率，为提升程序性能提供了帮助。反之，访问没有对齐的内存空间，CPU访问的最小单元必须以Byte为单位进行，且容易发生上述跨内存块访问导致的慢速情况。此外如果对没有对齐的数据使用存储长度不相同的强制类型转换，则会导致访问到其他变量的存储空间，这使得程序的鲁棒性以及灵活性远远不如使用了变量内存对齐的同种实现。