# 实验1 Cache替换策略设计与分析

计54 2015011340 骆轩源

## 【已有的替换策略】

- **LRU策略**:

  假如访问的内容有**时间局部性**，也就是最近访问过的内容，在短时间内再次被访问的可能性很大，基于这个假设，LRU维护了一个按照最近访问时间排序的栈(或者说队列)，栈底是最新访问的内容，栈顶是最长时间没有别访问的内容， 每次需要替换的时候，会把栈顶(也就是最近最久未使用的内容)给替换掉，每次访问的时候，会把访问的元素移动到栈底。 如果不用任何数据结构，LRU的时间复杂度是$\Theta(n)$，其中n是assoc大小，如果用平衡树来维护的话，时间复杂度就是$\Theta(\log n)$，由于assoc不大，也就16左右，所以$\Theta(n)$也能接受。

- **RANDOM策略**:

  简单来说就是每次随机选择要被替换的元素，这样做看起来很蠢，其实能防构造数据，比如序列a b c d e a b c d e....，就可以把assoc = 4的 LRU 卡的 Miss Rate接近$100\%$，但是随机就很难卡。

- **FIFO策略**:

  先进先出策略，这个策略也是假设访问的内容具有**时间局部性**，但是利用该性质的方法有些不同，LRU考虑的时间是最近被访问的时间，而FIFO考虑的时间是第一次进入缓存的时间。 故FIFO维护了一个队列，队首指向最早进入Cache的内容，队尾指向最晚进入Cache的内容，每次替换的时候，直接把队首替换掉就可以，每次Cache假如新的元素，就往队尾加，所以操作的时间复杂度是O(1)的。

## 【自己的算法】

我在本实验中，一共测试了两个自己的设计的简单的算法:

- **FreqSample**： 这个算法其实和 `最不常用算法` 有些类似，Cache需要对于每个Index维护其每一路(way)中内容的访问次数，如果是最不常用算法，就会在miss的时候，把访问次数最少的给替换掉。 那么对于FreqSample来说，是按照访问的次数，给每一路计算一个概率，然后按照这个概率选出应该被替换的元素，当然，访问次数最少的概率就会小，具体地，假设一共有$k$个way，其中他们的访问次数分别为$a_1, a_2, \ldots, a_k$， 令$M = \max_i\{a_i\}$，$sum = \sum_i (M - a_i + 1)$ ，则可以构造$P_i = \frac{M-a_i+1}{sum}$,不难验证$\sum_i P_i = 1$，我们就认为$P_i$是第$i_{th}$ way被换掉的概率。 注意到，这样设计下，当$a_i$小的时候，$P_i$大，并且没有哪个way的概率是0。 具体的代码实现如下:

```
INT32 CACHE_REPLACEMENT_STATE::Get_Freq_Victim( UINT32 setIndex )
{
    // Get pointer to replacement state of current set
    LINE_REPLACEMENT_STATE *replSet = repl[ setIndex ];

    UINT32   freqWay   = 0, sum = 0, pos,L,R,Max = 0;
    //replSet[i].now 存了a_i,也即访问次数
    //如下代码在计算最大值Max
    for (UINT32 way = 0; way < assoc; way++)
        Max = replSet[way].now > Max ? replSet[way].now : Max;
    //如下代码在计算sum
    for (UINT32 way = 0; way < assoc; way++) sum += Max - replSet[way].now + 1;
```

```
    //要保证选出way的概率是 P[way] = (Max - replSet[way] + 1)/sum
    //我采取的方法是，假设有一条长度为sum的线段
    //对于每个way给它分配一个长度为P[way]*sum的区间
    //然后随机一个位置pos，pos 均匀落在[0,sum-1]之间，看落在了那个way的区间里
    pos = (rand() % sum);
    L = 0;
    for (UINT32 way = 0; way < assoc; way++)
    {
        R = L + Max - replSet[way].now + 1;
        if (pos >= L && pos < R)
        {
            freqWay = way; break;
        }
        L = R;
    }
    // return freq way
    return freqWay;
}
```

为什么要这么设计，我当时的想法是，因为最不常用算法是一个确定算法，然而在Cache替换这个问题上，其实并没有一个固定的策略能做到最好，我们统计访问次数，**无非是依据过去的统计数据，来企图逼近未来的情况**，两个访问次数接近的way，比如一个1000次，一个999次，不能绝对认为那个999次以后的访问次数一定小于那个1000次的，因为他们非常接近，所以他们应该要以差不多的概率被换出去。 在我的设计里， 两个way的被换出的概率$P_i - P_j$，直接和$a_i - a_j$成正比。

- **Mixup**： 为什么要试一试mixup呢，是因为，我实验发现 FreqSample在平均情况下缺失率高于LRU，但有些程序下又能比LRU好，所以我就想了一个简单的方法，每一步替换的时候，我以0.65的概率使用LRU策略，以0.35的概率使用FreqSample，做了这样一个小改动之后，平均起来比LRU要好了。

```
INT32 CACHE_REPLACEMENT_STATE::Get_Freq_Victim( UINT32 setIndex )
{
    // Get pointer to replacement state of current set
    LINE_REPLACEMENT_STATE *replSet = repl[ setIndex ];

    //用freqSample策略计算一遍
    UINT32   freqWay   = 0, sum = 0, pos,L,R,Max = 0;
    for (UINT32 way = 0; way < assoc; way++) Max = replSet[way].now > Max ?
replSet[way].now : Max;
    for (UINT32 way = 0; way < assoc; way++) sum += Max - replSet[way].now + 1;

    pos = (rand() % sum);
    L = 0;
    for (UINT32 way = 0; way < assoc; way++)
    {
        R = L + Max - replSet[way].now + 1;
        if (pos >= L && pos < R)
        {
            freqWay = way; break;
        }
        L = R;
    }
```

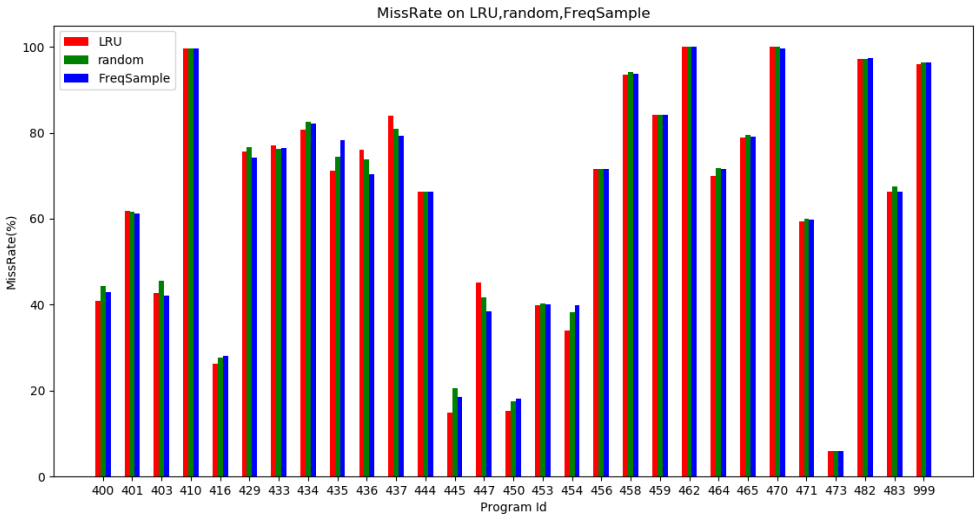
```
    //用LRU策略计算一遍
    int lruWay =  Get_LRU_Victim(setIndex);
    //mixup: 0.65的概率走LRU, 0.35的概率走freqSample
    if (rand() %100 < 65) return lruWay;
    return freqWay;
}
```

## 【实验结果】

一共做了两组对比实验，第一组对比了LRU/random/FreqSample的Miss Rate和CPI，第二组对比了LRU/random/mixup的Miss Rate和CPI。

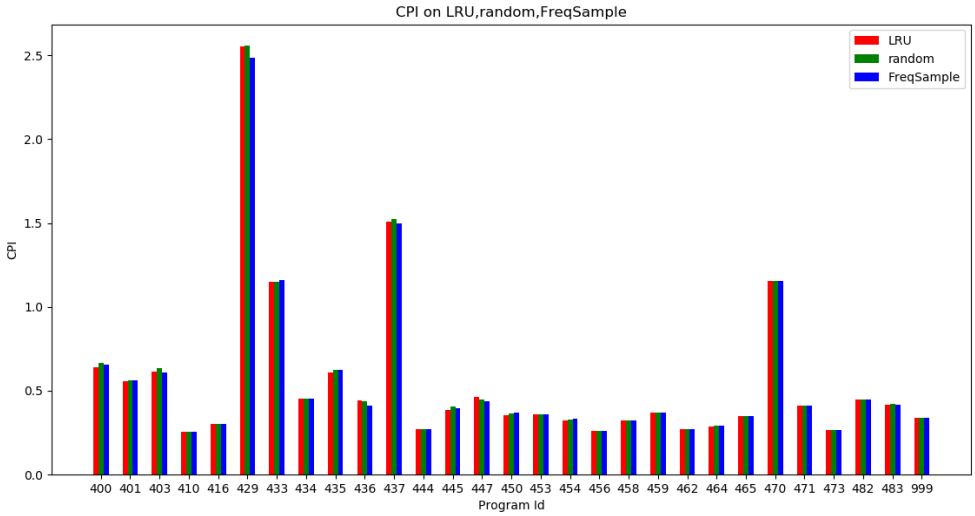
- **LRU && random && FreqSample**
  - **Miss Rate**

    FreqSample的Miss Rate比LRU平均高出$0.24\%$，比random平均低$0.51\%$, 观察下图可以发现，在某些程序下(447,437,433)，FreqSample 比LRU的Miss Rate好很多



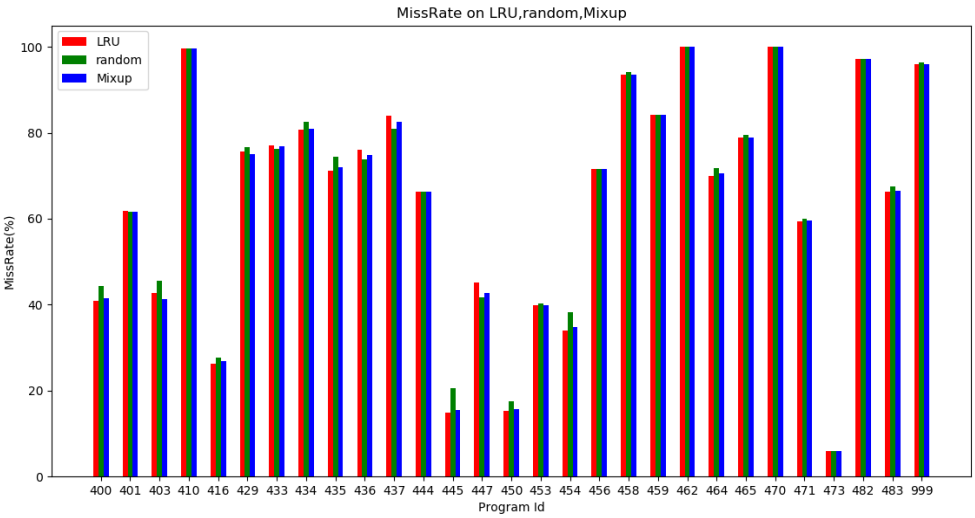

  - **CPI**

    FreqSample的CPI比LRU平均**低**0.00145，比random平均低$0.00568$, 在某些程序，比如436,429上明显低于LRU



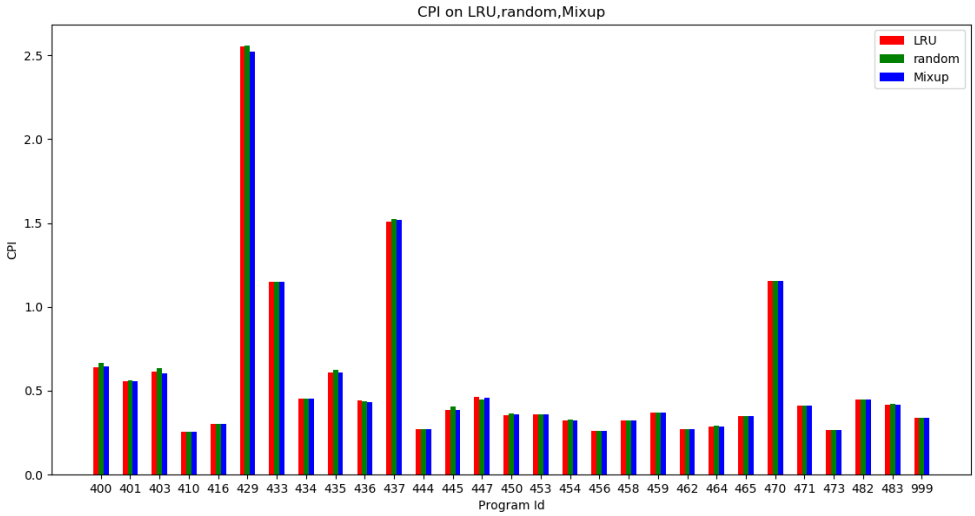

- **LRU && random && mixup**
  - **Miss Rate**

    mixup的Miss Rate比LRU平均**低**$0.1055\%$，比Random平均**低** $0.8657\%$观察下图可以发现，Mixup的Miss rate和LRU一般情况下都差不多，只是在有些程序，比如403,437上，略胜一筹。




  - **CPI**

    mixup的CPI比LRU平均**低**0.000935，比Random平均**低**0.0051703,稍微不如freqSample




**实验结果分析：**

通过实验对比可以发现几个现象：

- FreqSample 在有些程序上明显好于LRU和random，但是某些程序上又不如LRU，说明该策略只是对某一类特定的程序有优化效果
- FreqSample虽然miss rate略高于LRU，但是其CPI是低于LRU的，所以直接从使用上来看，应该是比LRU好。
- Mixup 能一定程度上综合LRU和FreqSample的长处，做到比二者都要优

**FreqSample适用范围的分析：**

我们的FreqSample本质上是统计了，每个way在历史中被访问的频率，一定程度上，能反应每个way被访问的概率，我们基于这个被访问的概率，构造一个它们不被访问的概率，然后以这个概率投掷一次骰子，选出应该换出的way。

- **顺序访问**：

  这样一种策略对于顺序访问都是没有优势的，因为顺序访问很难用到重复的元素。

- **循环访问**：

  循环访问的话，假如访问的序列是a b c d e a b c d e ....，然后assoc = 4，那么按照LRU, 在第一次e缺失的时候，都会把a换掉，而下一个刚好要访问a，所以又缺失了，依次类推，之后每次都会缺失。 而对于FreqSample来说，e缺失的时候，只有1/4的概率会换掉a，所以不会每次都缺页。 在这种情况下，random也可以做的跟FreqSample一样的效果。

- **分支跳转**：

  假如一个条件语句有2个分支，每个分支被走的概率是$P_1$和$P_2$，那么$P_i$能很好的被历史的统计信息给得到，按照LRU很难刻画这样的事情，LRU只考虑最近访问的是哪个分支就把哪个分支存下来，这个对于简单的for循环边界的分支，或者while循环边界的分支（也即走两边的概率很不平衡）是非常有利的。 但是对于真正复杂的分支跳转，相邻两次跳转就不一定是跳到同一个地方。 这两种情况分别可以用代码来看看：

```
x = 0;
//考虑这一个分支跳转，前1000次都是跳到同一个地方，只有最后一次才跳出去
//故对LRU非常有利
while (x <= 1000)
{
    x += 1;
}
```

```
//这个例子中，跳A和跳B的概率差不多，所以只有估计到P_1和P_2
//才能做一个明智的判断，这种情况下FreqSample比较有优势
if (rand() %100 < 40)
    do_A();
else do_B();
```

Random算法，则是在每个分支的概率相同的时候，能做到不差的结果。

## 【实验总结】

通过这个实验一方面深入了解了各种Cache的替换策略，学习了他们的中心思想，更重要的是，由于每个Cache替换策略都有其擅长应对的情况，所以我们在写程序的时候，要了解CPU的Cache替换策略，尽量迎合该策略的长处，使得自己的程序运行起来缺失次数更少，也就使得速度更快。