2016~2017春季学期计算机系统结构

实验 *1 Cache* 替换策略设计与分析

实验报告

计43

唐玉涵

2014011328

2016~2017春季学期计算机系统结构

实验 *1 Cache* 替换策略设计与分析

实验报告

1、任务说明

1. 理解学习 LRU 及其它已经提出的 Cache 替换策略
2. 在提供的模拟器上实现自己设计的 Cache 替换策略
3. 通过 benchmark 测试比较不同的 Cache 替换策略
4. 在实验报告中简要说明不同 Cache 替换策略的核心思想和算法
5. 在实验报告中说明自己是怎样对不同的 Cache 替换策略进行测试的
6. 在实验报告中分析不同替换策略下，程序的运行时间、Cache 命中率受到的影响

2、不同 Cache 替换策略的核心思想和算法

1. LRU替换策略

LRU（Least Recently Used，近期最少使用）算法是把CPU近期最少使用的块替换出去。这种替换方法需要随时记录Cache中各块的使用情况，以便确定哪个块是近期最少使用的块。每块也设置一个计数器，Cache每命中一次，命中块计数器清零，其他各块计数器增1。当需要替换时，将计数值最大的块换出。

2. LFU替换策略

LFU（Least Frequently Used，最不经常使用）算法将一段时间内被访问次数最少的那个块替换出去。每块设置一个计数器，从0开始计数，每访问一次，被访块的计数器就增1。当需要替换时，将计数值最小的块换出，同时将所有块的计数器都清零。

这种算法将计数周期限定在对这些特定块两次替换之间的间隔时间内，不能严格反映近期访问情况，新调入的块很容易被替换出去。

3. 随机替换

最简单的替换算法是随机替换。随机替换算法完全不管Cache的情况，简单地根据一个随机数选择一块替换出去。随机替换算法在硬件上容易实现，且速度也比前两种算法快。缺点则是降低了命中率和Cache工作效率。

4. FIFO算法

FIFO（First in First out），先进先出。在FIFO Cache设计中，核心原则就是：如果一个数据最先进入缓存中，则应该最早淘汰掉。也就是说，当缓存满的时候，应当把最先进入缓存的数据给淘汰掉。

5. RRIP算法

核心思想为将访问间隔较小的块留在cache中，从而提高命中率。维护Mbits记录PPRV，选择PPRV值为2的M次方减一的项替换出去，如果没有该项，则每项PPRV值+1，之后重复扫描。新进块PPRV置为2的m次方-2，命中块清零。采用此种方法描述访问间隔并通过访问间隔来预测访问概率。

3、自己实现的 Cache 替换策略

我实现的Cache替换策略为时钟替换策略，即FIFO策略与LRU替换策略的综合。先进先出替换策略完全不考虑过去的做法，而LRU考虑的时间较长，时钟替换策略既考虑过去，考虑的时间又不是那么长。时钟替换策略的思路是对页面的访问情况进行大致统计。

实现思路为：定义页表项，其中一位访问位描述过去一段时间的访问情况，一位指向cache。将页表组成环形页表，指针指向最先调用的页面。

具体的实现方式为定义环形页表，指针指向初始位置，当命中时，将对应页表中的访问位定义为真。当缺失时，从指针位置开始进行判断，如果当前位置访问位为真，将访问位置为假，指针指向下一位，直到指针指向位置的访问位为假，则当前位置为要进行替换的位置，并将指针指向下一位。

4、测试方法及结果

为了对三种不同的Cache替换策略进行性能测试及对比，我使用python编写了测试代码自动运行模拟器，并输出测试结果（缺失率），以下给出针对我自己实现的时钟替换策略的测试代码：

```
import os
```

```
filelist = os.listdir("traces")
num = 2
os.system("mkdir "+str(num))
for file in filelist:
    if file.endswith("out.trace.gz"):
        try:
            command = "time ./bin/CMPsim.usetrace.64 -threads 1 -t "+"./traces/"+file+" -o ./"+str(num)
+"/"+file[:-13]+".stats -cache UL3:1024:64:16 -LLCrepl "+str(num)
            print '\n'
            print command
            print '\n'
            os.system(command)
        except:
            pass
```

对于其他两个Cache替换策略，只需修改对应的replacement_state.cpp文件即可。

实验数据见下表：

缺失率（百分比）

编号	LRU算法	随机替换算法	时钟替换算法（自己实现）
400	40.7952	44.4057	41.1228
401	61.7369	61.6783	62.1271
403	42.7521	45.6397	43.5188
410	99.6634	99.6634	99.6634
416	26.3204	27.6546	26.5248
429	75.676	76.6152	76.7433
433	77.0164	76.2749	76.6981
434	80.81	82.6395	81.4494
435	71.1979	74.3719	71.7569
436	76.1164	73.7824	77.3741
437	84.0284	80.9639	83.9634
444	66.2603	66.3123	66.2719
445	14.919	20.4838	17.1557
447	45.0892	41.7856	47.1231
450	15.2011	17.5536	15.246
453	39.9258	40.2017	39.9801
454	34.05	38.3193	34.1872

456	71.4736	71.4736	71.4868
458	93.5415	94.1036	94.0842
459	84.1525	84.1525	84.1525
462	100	100	100
464	70.0086	71.7721	70.1737
465	78.8694	79.4314	79.2599
470	99.9725	99.9385	99.9772
471	59.4443	60.0557	59.5862
473	5.84718	5.84718	5.84718
482	97.156	97.262	97.1913
483	66.2759	67.5795	66.6189
999	95.9373	96.3212	95.8733
Ls	96.9842	96.9842	96.9842
平均值	65.70738	66.44224	66.07138
总缺失/总查找数	77.3269	77.3969	77.7499

运行时间：随即替换算法 < 时钟替换算法 < LRU算法

5、实验结果分析

对比以上三种三种不同的Cache替换策略，逐一分析如下：

1、LRU替换策略

LRU法虽然比较好地反映了程序局部性规律，但是这种替换方法需要随时记录Cache中各块的使用情况，以便确定哪个块是近期最少使用的块。LRU算法实现起来比较复杂，系统开销较大。通常需要对每一块设置一个称为计数器的硬件或软件模块，用以记录其被使用的情况。这种算法保护了刚调入Cache的新数据块，故具有较高的命中率。由于需要维护将元素调往栈顶，其余元素一次往下的过程，所以会使得程序的运行时间变长。

2、随机替换策略

随机替换算法不考虑Cache块过去、现在及将来的使用情况，但是没有利用上层存储器使用的“历史信息”、没有根据访存的局部性原理，故不能提高Cache的命中率，命中率较低。而运行时间由于只有随机数发生器的影响，所以时间影响较小，运行时间短，实现速度快。

3、时钟替换策略

时钟置换算法在上一次的置换位置后，选择最早的最近使用较少的块替换出。这种方法由于没有完全的统计每个块的顺序，而是只统计了最近是否使用，顺序则是按照先进先出的方式，所以只反映了程序部分的局限性。时钟替换策略较为合理，是LRU与FIFO算法的综合。所以命中率有所提升，但是一般不会高于LRU算法。

同样由于时钟转换算法只有对应的标记位，所以在寻找置换位置时，可能会比LRU慢，可能比LRU快，但是由于有标志位存在，所以在update时，时钟置换策略比LRU快。总体而言，时钟置换策略的运行时间比随机替换策略长，比LRU置换策略短。

6、实验小结

在本次实验中，我深入学习了各种不同的 Cache 替换策略以及不同替换策略对程序运行性能的影响，而且动手实现自己的 Cache 替换策略（时钟置换算法）。从整体上看收获很大，但也由于各种原因未能按时提交，在此向助教大大真诚地道歉。在最后也十分感谢汪老师和助教大大一学期的陪伴，谢谢。