Cache 替换实验报告

王琛　计 65　2016011360

1　什么是 Cache 替换

Cache 是主存储器的高速缓存，当要访问的内容不再 Cache 中时，则需要将主存中相应的内容。由于容量有限，Cache 中的一块可能对应多块相应大小的物理地址，装入时可能有多个可被替换的 Cache 块。而 Cache 的组织方式一般分为三种：全相连，多路组相连和直接映射。实际应用中基本采用多路组相连的方式，则 Cache 替换就是在多路中按照某种策略选择一路进行替换。不同的替换策略对于访存性能有着不同的影响。

2　已有的不同 Cache 替换策略分析

2.1　Random

Random 算法就是在所有可供选择的 Cache 中随机选取一块进行替换。由于没有考虑程序的访存特征，Random 算法对于相同的程序效果并不稳定。其优点在于实现简单，并且对于不同的极端情况也能够很好的适应。

2.2　LRU

LRU 算法利用过去的访问行为推测未来，其基本思想是替换 Cache 块中最久没有被访问的。可以用类似堆栈的方式实现，每个块进行编号，当某个块被访问时，将所有小于该块的编号加 1，该块编号置 0；替换时选择编号最大的块。使用 LRU 算法，当程序展现出较好的局部性时，效果比较好。但是当程序执行扫描而 Cache 容量相对较小时，命中率比较低，因为前面的总是会被替换出去。

2.3 LFU

LFU 是根据近期访问的频率来确定被替换的块，频率最低的会被替换出去。LFU 也利用了程序的局部性特征，但是问题在于其容易替换掉刚刚被换进来的块。而且当程序切换到另一块访存空间时，原来的块由于访问次数多不容易被替换，导致刚切换很容易缺页。

2.4 Protected LRU

Protected LRU 同时考虑访问次数和访问的时间。由于 LRU 替换访问时间早的块，那些之前访问次数多最近没有被访问的块很容易被替换掉，而这些块接下来很可能再次访问。因此，每次替换时，选择一定数目访问次数较多的块，设置为 Protected，不会被置换；在剩下的块中选择最早被访问的替换。在本框架中，我进行了实现，每个 Cache 块记录访问次数，为了避免 LFU 的问题，最大次数设置为 8 次，每次替换时保护 12 个，从剩下的 4 个中选择最早使用的替换出去。

2.5 LIP

LIP 算法与 LRU 的最大区别在于，当新的 cache 块被换入时，LIP 将其放到 LRU 的位置，只有当再次被访问时，才将其放到 MRU 的位置。这样避免了新换入的块将原 cache 中的块挤出去。

2.6 BIP

BIP 思路和 LIP 差不多，是将新换进的块随机放到 MRU 和 LRU 的位置。使用一个参数 ϵ，当随机的数比 ϵ 大时，放到 LRU，否则放到 MRU。

2.7 RRC

RRC 算法也是基于 LIP 的思路，是 17 年新发表的一篇文章 [1]。当一个 cache 块被替换进来时，首先将其放到 LRU 的位置，只有当其被引用的次数到达某个阈值时将其挪到 MRU 的位置。如图所示：

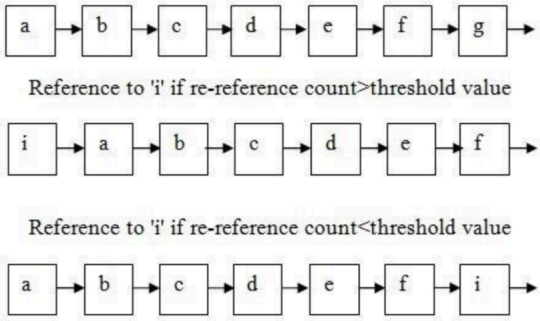
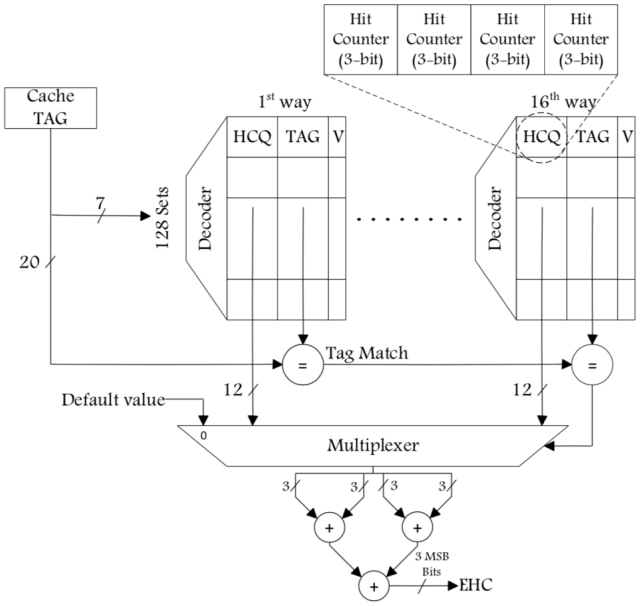


Fig. 2. New policy based on re-reference count

我在本次实验中实现了这个算法，但是效果似乎并不理想。

2.8 EHC

EHC 全称为 Expected Hit Count，也是 18 年提出的 [3]。该算法的核心是一个 cache 块被重新使用的次数和其两次被重用的距离有关系。算法将相邻的四个块作为整体，记录他们在一段时间内的使用次数；然后据此推断下一次访问的间隔。选择 victim 时，选择下一次访问间隔最远的块。



2.9 Score

Score 算法是 Cache replacement championship 中提出的算法 [2]，思路是给每个块一个 score，选择 victim 时从分数低于阈值的随机挑选。刚运行时，每个块的初始分数为最大分数的一半，每个阶段初始分数变化一个 step，初始变化方向为增加。监控每个阶段的缺失率，如果当前缺失率大于之前的阶段，则改变初始分数变化的方向。访问某个块时，该块的 score 增加，其余的块 score 减小。

3 新提出的 Cache 替换策略 – WMBP

我提出的 Cache 替换策略是 Workset Miss Rate based Points policy (WMBP)，是基于 Score 策略作出的改进。Score 算法的问题在于无论是初始 score 的变化量还是选取 victim 的阈值都是一个绝对量，无法反应换入换出块在队列中的相对位置。因此如果前一阶段剩余块和当前阶段的分数绝对值相差较大时会出现问题。我的思路是给每个块一个 point，根据一个 Workset 的缺失率确定初始 point 和 victim 的选择。下面具体阐述：

3.1 初始 point 的选择

Point 的相对大小其实反映了 LRU 算法中的相对位置。当换入一个新的块时，无论是直接放入 LRU 的位置还是 MRU 的位置，在特殊的情况下性能都会很糟糕。之前的算法，无论是 LRU, MRU, LIP, BIP 放入时都没有考虑程序之前的访问特性。WMBP 的思路是，如果当前工作集的缺失率越高，说明访存区域改变的可能性越大，因此新换入的块被用到的可能性更大，故应接近 MRU 的位置，即分数高；而当前如果缺失率很小，则新换进的块不适合放入 LRU 的位置，因为挤掉的块后续访问的可能性大，因此初始分数应该较低。具体实现时，找到队列中的 maxPoint 和 minPoint，将 minPoint+missRate*(maxPoint-minPoint) 作为初始分数。此外，只有当工作集窗口大小到达最大窗口的某个比例才开始使用这种方法，否则直接将初始 point 设置最大 point 一半。这样做是为了防止窗口较小时发生抖动从而影响结果。代码如下：

```
1    if(5*numAccess > WINDOW_SIZE) {
2        float missRate = float(curMiss)/float(numAccess);
```

```
            replSet[updateWayID].point = minPoint + int(missRate*(maxPoint-
                minPoint));
        } else {
            replSet[updateWayID].point = MAX_POINT >> 1;
        }
    }
```

3.2 cache 命中时数据的更新

当 cache 命中时，被命中的块 points 要增加，而没有被命中的块 point 相对要减小。

3.3 Victim 的选择

在 Score 算法中，Victim 是通过固定阈值过滤的，这样做的问题是阈值对结果影响较大。可能出现同一阈值有的情况下全部过滤，有的情况一个也无法过滤。因此，在 WMBP 算法中，Victim 有一个集合，从这个集合中挑选 Victim，而这个集合的大小是动态变化的。同样，当前工作集的缺失率决定了集合大小如何变化。缺失率越高，这个集合越大，更多的块都可能被替换掉；缺失率低，只有少数的 point 小的被替换。这个思路实际上借鉴了 PLRU，只是 PLRU 的 Victim 集合大小是固定的。实现时，将集合大小首先设置为定值，当缺失率大于一半时，增加集合大小；否则减小集合大小。同样和初始 point 选择一样，需要考虑当前窗口过小的问题。

```
    if(5*numAccess > WINDOW_SIZE) {
        float missRate = float(curMiss)/float(numAccess);
        if(missRate > 0.5) {
            victimNum += missRate * 4;
        } else {
            victimNum -= missRate * 4;
        }
        if(victimNum > VICTIM_SET_MAXSIZE) victimNum = VICTIM_SET_MAXSIZE;
        if(victimNum < 1) victimNum = 1;
    }
```

4 各种替换算法的比较

本次实验中，我实现了 PLRU, SCORE, RRC, WMBP 算法，框架中已有 LRU 和 Random。对这些算法进行了测试，其中 RRC 算法结果有

些离谱，结果不再列出。测试环境为 Ubuntu 10.04，使用了提供的 29 个 benchmark，且测试时外部环境相同，没有其他运行的程序（实验证明这点对时间的测量影响较大）。Cache 大小均为 1024KB，16 路组相连，每个块大小为 64B。

每个算法在不同 benchmark 下的缺失率和运行时间见下图：

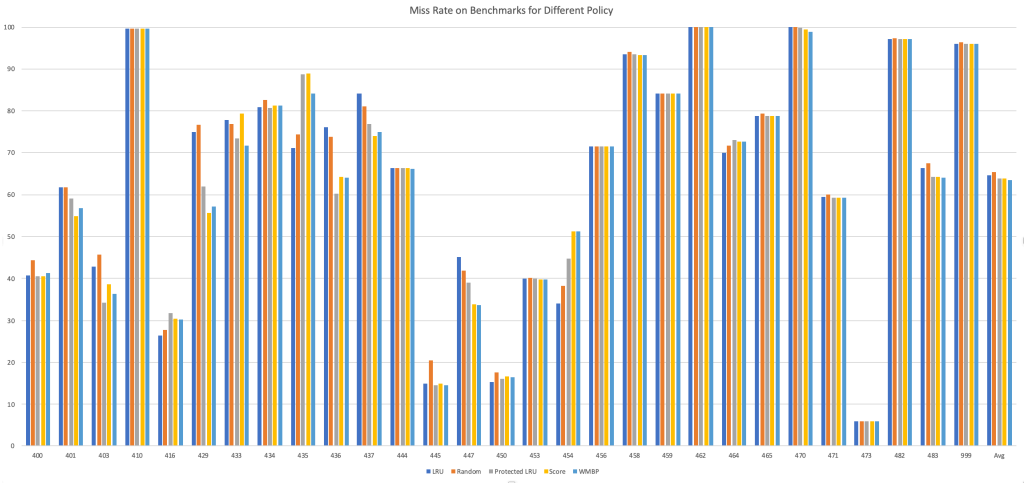


图 1: 不同算法的缺失率

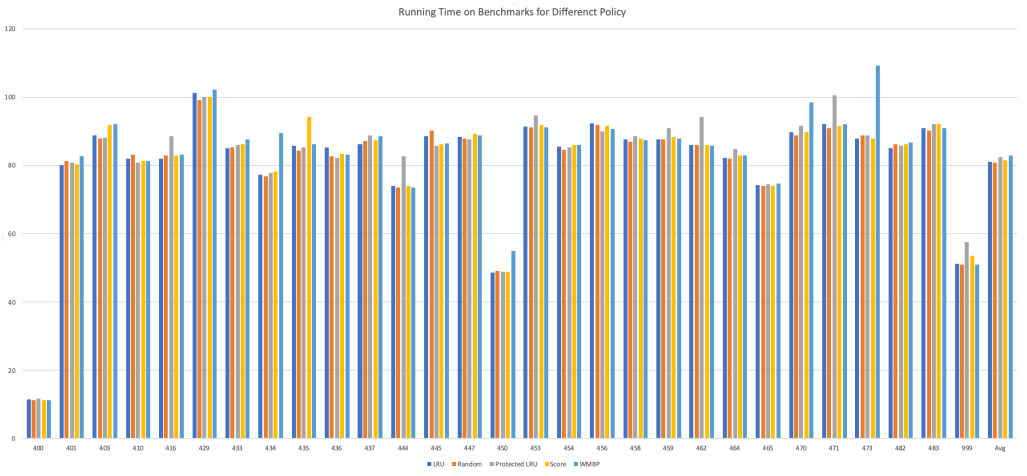


图 2: 不同算法的运行时间

分别比较 WMBP 和 Score、LRU 的缺失率：

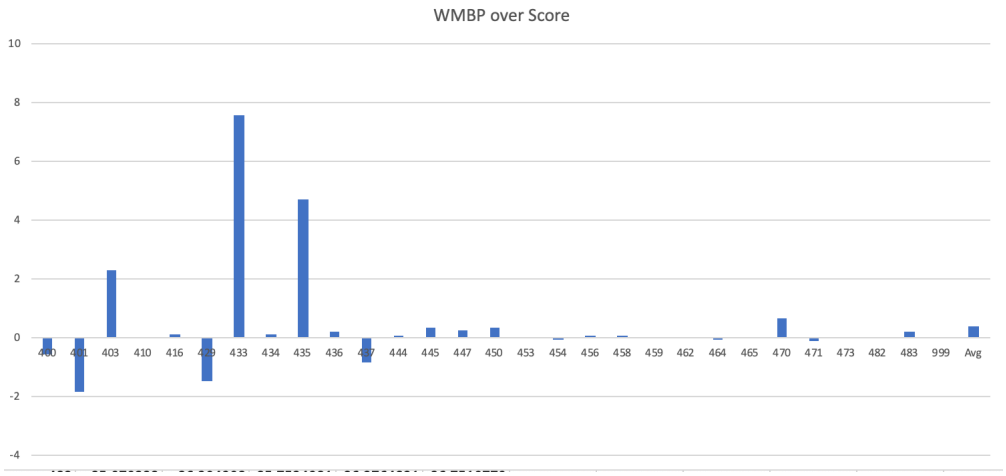


图 3: WMBP over Score

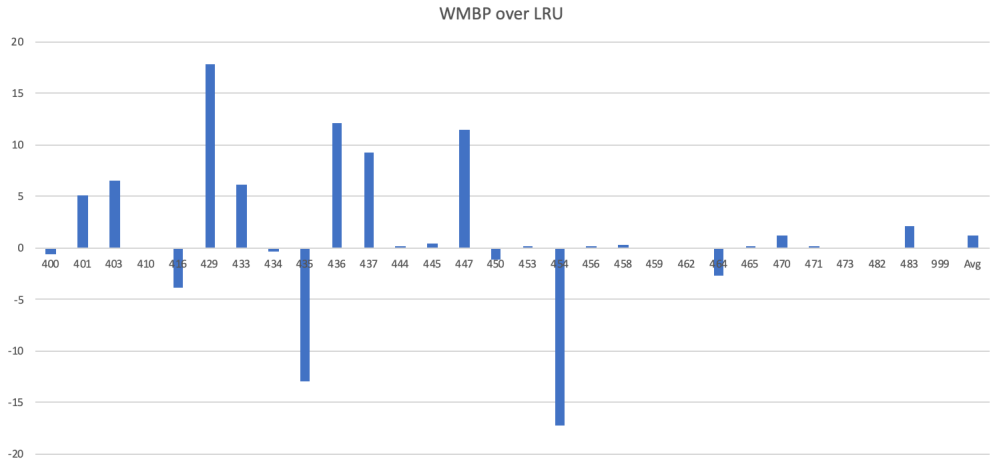


图 4: WMBP over LRU

结果分析

从图中可以看出，不同的算法和 baseline 相比时，有的 trace 较好，有的 trace 较差。在大部分的算法下，缺失率 WMBP < Score < PLRU < LRU < Random；平均下来也是如此。有些情况值得注意：

- 416/435/454/464 中 Random 算法比除了 LRU 之外的算法都要好。说明这些 trace 更加接近 LRU 的访存特征。
- 433/436/437/447 中 Random 算法都要好于 LRU 算法，说明这些 trace 可能会继续使用未被替换的块，而 LRU 会将这些块优先换出，因此效果不好。
- 410/462/482/499 这几个 trace 缺失率很高，所有的算法都难以改善；可能是因为程序几乎没有局部性，因此无法改善。
- 470 这个 trace 在 LRU/PLRU/Random 算法下缺失率大于 99.7，而 Score 将其降到了 99.4，WMBP 进一步降到了 98.8。这个 trace 应该是 cache 刷新较快，使用 LRU 和 PLRU 很快会被替换出去，而 Score 和 WMBP 算法由于采用记分机制，原有的 cache 块不容易被替换，因此较好。而 WMBP 初始分值和换出策略还考虑了相对顺序，会更合理一些。

而对于不同替换的运行时间，平均而言 WMBP > PLRU > Score > LRU > Random。和命中率一样，就平均而言其实相差并不大，但是在某些 trace 上相差比较明显。比如 PLRU 在 444 和 462 比其他要高很多；而 WMBP 在 450 和 473 高的很明显。这也是显而易见的，因为这两种算法都要首先选取一定大小的 victim set，所以会消耗较多的时间。我原本认为 WMBP 会比其他几种算法消耗比较多的时间，但是在有些情况下时间消耗甚至比较少，我认为是局部性较好，从而 victim set 比较小，比如 999，消耗时间较少而且准确率还比较高。

自己的 trace

我认为我设计的 cache 算法核心仍然是将新换入的块放到队列中的位置，LRU 是放在 MRU 出，MRU 是放在 LRU 处，BIP 随机放置，而 WMBP 是根据缺失率动态调整位置。所以，假设 cache 只有 4 路，一般的 LRU 碰到 abcdeabcde 这样重复的访存序列，缺失率会趋向 100%，因为总是会将下一个要访问的块挤出去。类似这种序列在现有的几种算法（除了 WMBP）效果都不好。而 WMBP 虽然不像 MRU 直接放到 LRU 位置，但是能动态适应，因此缺失率最低。比如下面的程序：

```
1    #include<iostream>
```

```
using namespace std;
int main(){
    int a[278528], i, j;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 278528; j++) {
            a[j] = i + j;
        }
    }
    return 0;
}
```

将其制作成 trace，结果如下：

Policy	Miss Rate	Running Time
WMBP	30.74	25.49
Random	34.05	24.31
PLRU	51.99	26.71
LRU	99.83	28.85
Score	32.66	31.03

这段程序是专门针对本次实验所采用的 cache 设计的，循环访问 a 数组，a 数组的大小正好比 cache 多一路。如果采用 LRU 算法，每次都会缺失；而 PLRU 会保护刚刚访问的，由于每个 block 有 8 个 int，很快被再次访问，因此缺失率将近一半；这种情况 Random 反而效果较好，因为不容易将下次要访问的挤出去；而 Score 和 WMBP 使用计分算法，而 WMBP 还考虑了缺失率和相对位置，因此缺失率最低；时间也仅次于 Random。

WMBP 算法优缺点分析

WMBP 的优点是显而易见的，克服了 LRU 一类算法可能出现的问题，并且在 LRU 表现较好的情况下效果仍然很好。由于将计分策略和缺失率相结合，算法能够比较好的随着程序访存特性变化，因此总体的缺失率也比较低。

当然，WMBP 实现有些复杂，需要记录较多的变量，而且涉及到浮点运算，所以运行时间较长，不过仍然在可接受范围之内。另外，此算法在片上可能难以实现，只能停留在模拟器阶段。

5 实验总结

本次实验，我对于不同的 cache 替换策略有了一个初步的认识，也阅读了竞赛的论文和近年的几篇论文。虽然 cache 替换策略并不是一个新的话题，但是仍然保持着一定的活力。另外，在实验中，我也体会到了没有一个完美的 cache 替换策略，每种方法都有不适用的情形。

参考文献

[1] Sreedharan, S., & Asokan, S. (2017, March). A cache replacement policy based on re-reference count. In 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT) (pp. 129-134). IEEE.

[2] Nam, D., Rosario C., etc. SCORE: A Score-Based Memory Cache Replacement Policy

[3] Vakil-Ghahani, A., Mahdizadeh-Shahri, S., Lotfi-Namin, M. R., Bakhshalipour, M., Lotfi-Kamran, P., & Sarbazi-Azad, H. (2018). Cache replacement policy based on expected hit count. IEEE Computer Architecture Letters, 17(1), 64-67.