# 计算机系统结构实验一报告

计算机系　计 43　2014011330　黄家晖

April 27, 2017

## 1　不同 Cache 替换策略分析

### 1.1　随机算法

随机算法实现容易，在每次缺失的时候仅仅需要取随机数作为替换的路即可。这种方式没有考虑到访问的模式，同时也没有考虑到访问的特性，对于相同的程序不具有稳定性。该算法的一个好处是对于各种情况的访问模式具有一定的免疫性。

### 1.2　LRU 算法

LRU 算法 (True LRU) 通过过去的访问模式估计未来的访问模式，每次需要替换的时候，选择最长未被使用的行进行替换，如果程序具有较好的局部特性，且 Cache 容量足够大，则 LRU 的命中率较为可观。但是，如果程序有类似于**扫描**的访问模式，且扫描长度大于 Cache 可接受容量，则 LRU 算法总会替换最早进入的行，导致缺失率很高。一般地来说，一个新换入的行，如果之后再也没有访问过，在 LRU 算法框架下，该页需要很长时间才会被替换出去，占用 Cache 资源。

### 1.3　LIP 算法

针对 LRU 的相关问题，一类更广泛的 Protected LRU 算法提出将过滤列表和重用列表进行分离。LIP 算法就是这样的一个例子，与 LRU 恰恰相反，LIP 算法选择替换刚刚被使用的行。这就使得 LIP 在出现上述问题的时候能够保留若干个开始访问时的行，命中率高于 True LRU 算法。

LIP 算法的问题也很明显，一些行有可能会常驻 Cache（除非被重用）。人们为了融合二者的优点，设计了 BIP 算法或是 DIP 算法来综合这两种算法。

## 1.4 2Q 算法

除了通过 LIP 的方法解决 True LRU 的问题，人们还设计了 LRU-K 算法解决这个问题，这个算法的核心思想是统计并只将最近使用了 K 次的行放入 Cache，否则直接从内存读取。2Q 算法就是 K=2 时的一个实例变种，它通过在 Cache 中同时维护一个 FIFO 队列和一个 LRU 队列，每次有访问的时候，先将行放入 FIFO，如果再次访问才移动到 LRU 队列中执行 LRU 算法。可以设想，如果一个行只被访问了一次，那么就会很快跟随 FIFO 队列退出 Cache。

显然，使用 2Q 算法需要占用一部分宝贵的缓存空间来专门维护 FIFO 队列，且实现起来比较复杂，实际应用效率不高。

## 1.5 Score 算法

Score 算法是 Cache Replacement Championship 中提出的替换算法之一，作者是美国的 N. Duong, et al. [1]。其核心思想是给予每路一个单独的分数，每次选择分数最低的几个路随机选择进行替换。另外，如果有命中或是不命中的情况出现，需要对不命中的块进行减分，对命中的块进行加分。新换入的行会被给予一个初始分数，这个初始分数是动态变化的，如果某一次进行变化之后命中率降低，则更改变换的方向，否则初始分数保持不变。

相比于 LIP 算法和 2Q 算法来说，该算法没有直接使用队列的形式来维护，而是通过分数值来间接反映在队列中的位置。在 Score 算法中，每一个新进入的行的分数值是动态变化的，而不是最高值，这就说明如果初始分数选择恰当，新进入的行也可能不会存在 Cache 过长的时间，同时分数的存储也不会占用过多的硬件资源，告别了 LIP 和 2Q 的缺点。

因此，初始分数的选取以及根据分数的淘汰策略对于 Score 算法来说很重要，我自己提出的算法就是对这两方面进行了改进。

## 1.6 自己提出的 Token 算法

Token 的意思在这里与 Score 很相近，只是一个名义上的叫法而已。但是算法本身与 Score 相比有了一些改进。算法描述如下：

**初始化** 定义淘汰置信系数 $t$，初始 Token 为 $m$，监控窗口大小为 $w$。

**访问 Cache 时** 如果某一路命中，则增加这一路的 Token，并减小其他路的 Token。如果不命中，则为新换入的行置为 $m$。另外，监控本次的 $w$ 条指令与上次 $w$ 条指令的缺失率变化，假设缺失率变化为 $\Delta k$，如果 $|\Delta k| > threshold$，则减小监控窗口 $w$，否则增加监控窗口 $w$（但不能超过

[1]SCORE: A Score-Based Memory Cache Replacement Policy, N. Duong, R. Cammarota, D. Zhao, T. Kim, and A. Veidenbaum (UC-Irvine, USA)

最大值)。如果缺失率相对于上个监控窗口增大（$\Delta k > 0$），则降低淘汰置信系数 $t$，否则增加 $t$，其余部分同 Score 算法，调整 $m$ 向使得缺失率增大的反方向变化。这一步的 $m$ 变化实际上基于对之前程序的**观察和学习**，从而能够使算法适应程序。

**选择替换的时候** 首先根据 $t$ 确定候选集合大小，如果置信系数比较低，则候选集合大小 $n$ 很大，反之候选集合大小 $n$ 很小。接着，对所有的分数进行排序，选择前 $n$ 小的，使用随机算法挑选一个进行替换。

可以看出，监控窗口的动态变化能够更加细粒度地监控缺失率变化情况，自动将窗口调整到合适的大小；而如果有连续的缺失，则会使得对于当前替换集置信系数较低，从而会进行更偏向于随机的选择；反之则会对分数的评定的准确性更加确信，更可能选择分数最低的进行替换。

# 2 Cache 替换策略测试与评价方式

针对 Cache 的替换方式的评价，本实验采用缺失率（Miss Rate）来衡量替换方式的好坏，以运行时间来衡量算法复杂度。虽然对于硬件 Cache，该算法可能会通过更并行的方式实现，运行时间不见得能够客观体现软件运行时间，但是可以从一定角度反映硬件系统设计的复杂度。

测试的时候使用提供的模拟器来进行程序模拟，`traces` 文件来源于网络学堂提供的实例，Cache 的构造是 1024KB，每行 64B，16 路组相连，第一级和第二级 Cache 使用 LRU 的策略，而第三级 Cache 使用自定义的策略。模拟器运行在虚拟 vagrant 环境中（系统为 Ubuntu 10.04 Server），使用 Python 进行计时和批处理。

# 3 各种替换策略的比较

LRU 算法、随机算法和 Token 算法的命中率对比和运行时间对比如图 1和图 2所示。

首先观察图 1缺失率，可以看出，使用了 Token 算法的 Cache 替换策略有一部分优于 LRU 算法，一部分不及 LRU 算法，而基本都优于随机替换算法。平均而言，Token 算法比 LRU 算法的准确率高出 0.98% 左右。然而，就一部分缺失率很高的程序（例如 410，462，缺失率接近 100%），Token 算法也不能解决这个问题，就个人推测可能这类程序属于顺序访问类程序，基本不会重复访问内存区域，导致每次 Cache 均缺失。

而就程序运行时间来说，Token 算法的运行时间比随机和 LRU 均要长。诚然，Token 算法为了捕捉程序运行情况，需要记录许多额外的变量，增加许多逻辑。这种运行时间上的长实际上代表着硬件实现的复杂性和可能

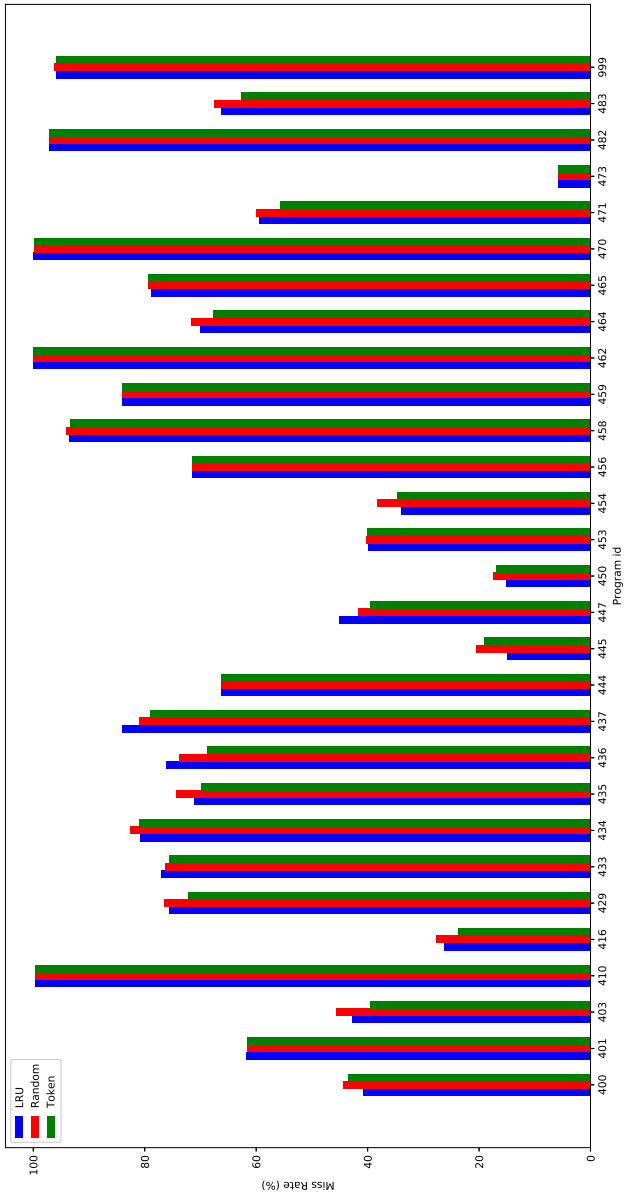


Figure 1: 不同策略命中率的对比

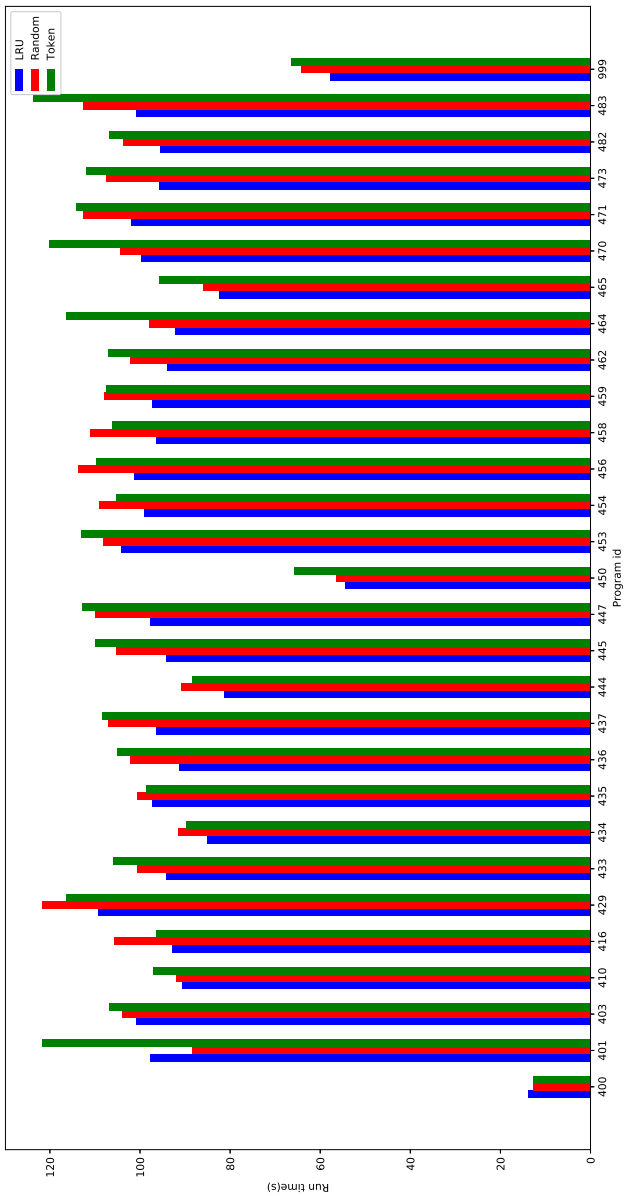


Figure 2: 不同策略运行时间的对比

功耗的增加。因此，在实际硬件设计的过程中，还应该仔细斟酌，平衡选择这些算法。

# 4 实验总结

本次实验中，我理解和学习了 LRU 及其他一些已有的替换策略，并尝试在模拟器上实现了自己的替换策略，实验证明，该策略相比 LRU 来说缺失率相近，相比随机算法来说缺失率降低，在某些数据上表现比较好。通过实际的测量，我对 Cache 替换的相关问题理解更加深刻了。