# Cache 替换策略实验报告

计72 马川 2017011409

## 算法简介

在本次实验中，我参考了LRU-K[1], 2Q[2] 以及 LIRS[3] 三个页替换算法的设计思路，设计了一个类似 LIRS 的 cache 替换算法，称为 PLRU 算法。该算法可以视为 LIRS 算法针对 cache 替换的一个简化实现，针对小部分高重复访问数据具有良好的适应性。该算法对 LIRS 算法进行了部分改动，大幅地减小了其占用的内存空间，使其可以正常地适用于局部 cache 替换。

### PLRU 算法

PLRU 代表 Priviledge LRU，即带特权位的 LRU 算法。该算法在 LRU 算法之上增加一位特权位(Priviledge Bit)，将同一相连组下的所有路的 cache 块分为具有特权与不具有特权两个类别，使得具有特权的 cache line 更难被替换。 PLRU 算法参考了 LIRS 的替换策略，并且结合了 LRU-2 和 2Q 的特点进行 victim 的选择以及状态位的维护。

#### 算法相关概念

##### 特权位

对于每一个 cache 数据块，其均具有一个 bit 位表示该块是否具有特权。所有具有特权位的数据块被认为会被频繁访问，不会被作为 victim 替换，只有不具有特权的块有被替换的可能。

##### 特权栈与等待队列

所有具有特权的 cache 块被视为在一个栈中维护，这个栈被称为“特权栈”。位于特权栈的元素一般情况下不会被替换，这样可以既可以充分利用程序的空间局部性，储存一段时间内被频繁访问的数据。还可以防止在一个大工作集上非集中分布的数据占用频繁访问的数据空间从而导致缺失率的上升。

所有不具有特权的块则视为在一个队列中维护，这个队列被称之为“等待队列”。位于等待队列中的元素被视为需要很长时间才会被再次访问，而这些块可以用来储存临时数据或是访问频率很低的数据。等待队列根据先进先出的原则决定下一次缺失时被替换的块。

特权栈和等待队列的长度之和永远不超过组相联的路数，且等待队列具有长度的上限和下限，其作用为：

1. 等待队列长度的上限使得用于储存临时数据的块数保持在当前相连数的比较低的比例当中，这样可以防止过多的块被用于储存临时数据从而挤占频繁访问数据的储存空间
2. 等待队列长度的下限保证了留出一部分块用于储存临时数据。下限的增加可以使得工作集的切换更加顺利。

下图示意了在程序运行了一段时间后，一个 cache 行中所有路的 cache 块所构成的特权栈与等待队列。图示中，每个块中包含一个三元组(x,y,z)，x表示其特权位，y代表其属性值的值, z表示块的编号。在本文后续的描述中，特权栈称为 S, 等待队列称为 Q。

Q

S

Llirs = 3

Lhirs = 2

(1,1,0)

(1,3,1)

(1,1,3)

(1,2,4)

(1,4,2)

）

一般来说，Q 的最小长度约为组相联路数的1%,Q的最长长度为组相联路数。

##### 属性值

每一个块使用 31 bits 作为其属性值（Property）。属性值会因为特权位的不同而表示不同的含义。

当一个块的特权位为1时，这个块被视为在特权栈中，其属性值代表直到这个块上一次被访问时访问过的所有不同块的数量。这个属性值参考了 LIRS 算法中 IRR [3]的设计，相对 LRU 而言，这种统计方法可以防止 LRU 仅对访问间隔进行统计的局限性，比如说针对一个大工作集上的循环访问。属性值越大意味着这个块可能与当前工作集距离越大。可以认为，所有位于特权栈中的块访问比较频繁，其 IRR 值较小，称之为 LIR 块即 Low IRR ；所有位于等待队列中的块具有较大的 IRR 值，称之为 HIR 块。

当一个块的特权位为0时，这个块被视为在等待队列中，其属性值代表其在队列中的位置，数值越大意味着其越靠近队头。根据等待队列的工作机制，所有没有特权的块按照先进先出的顺序储存和释放数据，属性值代表其在队列的位置，每此只需要维护每个块的属性值即可得出队列中各元素的相对位置。

此时有一个特殊情况：如果一个块的属性值为0，那么无论其特权位是否为1，其被视为既不在特权栈，也不在等待队列中。这说明当前这个块并没有储存数据。

通过和特权栈共享同一个变量储存数据可以大幅减少所需要的额外空间。相对于 LIRS 算法而言，PLRU 算法对于每个内存块只需要额外的 32 bits空间，与 LRU 算法需要的额外空间相同，并且不会同 LIRS 算法一样在某些特殊的情况下产生依赖空间的大量增长。

### 算法过程描述

根据” Adaptive insertion policies for high performance caching”[4]中的描述，我将 PLRU 算法的核心分为三个部分：第一个为命中策略(hit policy)，代表命中后应该如何维护结构中的各部分数据；第二个为被替换块选择策略(victim selection policy)，说明了应该如何选取被替换块并且更新部分数据；第三个为插入策略(insertion policy)，介绍了在发生缺失并且插入数据时应该如何维护结构信息。

#### 命中策略

先假设程序运行了一段时间，此时S的大小达到其最大值。此时的S与Q的状态如下：

Q

S

Llirs = 3

Lhirs = 2

(1,4,2)

）

(1,3,1)

(0,1,3)

(0,2,4)

(1,1,0)

此时如果 cache 命中，则包含两种情况：

1. 所命中的 cache 块具有特权，位于 S 中。此时只需要将其移动至栈顶即可。将栈中所有的块的属性值进行对应的修改
2. 所命中的 cache 块不具有特权，位于 Q 中。此时，本算法认为其可以获得特权，将其特权位置为1，属性值清零，并且将S中所有元素的属性值加一。如果此时栈已经达到最大值，则将栈底元素的特权值置为0，属性值置零，并且令 Q 中所有元素的属性值加一。这表示将这个元素从 Q 中移除，并且置于 S 的栈顶，如果此时栈已经满了，则将栈底元素移出栈，并且插入队尾。

PLRU 的命中策略类似于 LRU-2 的命中策略，即一个在队列中的块需要再命中一次才能进入栈。这个策略使得栈中的元素的 IRR 不低于队列中元素的最大个数。但是其通过限制栈的大小，并且仅仅在队列中的元素需要提升特权级的时候才将栈中元素移除，这样最大程度上保证了特权栈中的元素不被替换，防止抖动的发生。

命中2号块后

Q

S

Llirs = 3

Lhirs = 2

(0,1,3)

(0,2,4)

(1,1,2)

）

(1,4,1)

(1,2,0)

命中4号块后

Q

S

Llirs = 3

Lhirs = 2

(1,1,4)

(1,2,2)

）

(0,4,1)

(1,3,0)

(0,2,3)

#### 被替换块选择策略

如果产生缺失，则将 Q 的栈顶元素作为被替换块。因为本算法要求 Q 中至少有一个元素，故必然可以在 Q 中找到一个元素。如果该缺失强制缺失，即此时当前行所有路的 cache 块均没有数据，则取出下标最大的路用于插入到 Q 的队尾，令其成为 Q 的第一个数据。

#### 插入策略

在缺失发生后，已经释放了被替换块的相关数据并且插入新的数据。因为此时被插入的数据一定是在上一个策略中选择的块，故将该块的属性值清零0，特权位置为0，并且将S与Q中所有元素的属性值加一。这代表向栈中压入这个元素并且将这个元素插入队尾。 最后执行衰退算法

##### 衰退算法

PLRU 算法最大的问题在于，当一个块进入特权栈之后，其很难降低其特权级。所以衰退算法是为了在特定的时期释放特权栈中的内容，防止仅访问了两次从而进入特权栈的数据妨碍之后到达但是访问频率更加频繁的数据被使用。

衰退算法可以采用多种不同的算法，在这里我采用的是基于 IRR 的 LRU 算法，即在下一次访问当前块之前，访问块的种类数量。如果这个属性值大于组相联的路数的100倍，则说明工作集已经切换且该 cache 块中的数据并无法被利用，故降低栈底元素的特权级，并且加入队尾。该常数为实验测试所得。

## 算法命中率理论分析

PLRU 的设计主要针对 LRU 在两个工作集上切换时命中率会下降。这里，我将对几种简单的数据访问方式[5] PLRU 与 LRU 的命中率分别进行分析。

#### 时近友好型访问模式(a)

形如上式的访问方法可以称为“时近友好型”的访问模式，其可以保证在一段区间内，最新获得的数据会被最先访问。此时，设 cache 的组相联数为 ，则使用 LRU 的理论命中率为 ，只有最后加载的 W 路数据可以命中。

PLRU算法在前 k 次访问时全部 miss，并且等待队列中包含到的元素。在访问接下来 W 个元素时会命中，此时 PLRU 算法退化为 LRU 算法，命中率为 。

#### 循环顺序访问模式(b)

形如上式的访问模式会导致 LRU 的命中率大大下降。因为 LRU 根据最近访问的时间进行判断，在该模式下命中率为0。又因为 PLRU 算法基于 LRU 算法，其命中率依然为0。

#### 两个工作集的切换访问模式(c)

形如上式的访问模式表现出一个程序在两个工作集上反复访问时的一种情况。可以认为 P 是不重复访问工作集的部分其他数据。

此时 LRU 算法给出的命中率可以根据模式(a)进行计算，约为。

PLRU算法在访问工作集b时会因为P的长度而产生不同的结果。如果P的长度超过100W，则此时 PLRU 可以舍弃特权栈中的数据，并且根据 LRU 的方式处理工作集 ，其命中率与 LRU 相同，约为。当Ｐ的长度在100W以下，则在进入工作集后特权栈中的数据需要一些时间释放，如果在这段时间内可以返回到工作集，则命中率为，此时命中率会下降。

#### 重复集中访问小部分数据

针对重复集中访问的小部分数据，LRU 算法会在访问P时丢弃这部分数据，命中率为。PLRU算法在特权栈中保存工作集后便可以长时间保存和访问这些数据，命中率约为。

## 算法适用环境分析

通过上述理论分析，可以发现PLRU 算法的特点是可以很好地保存部分经常访问的 cache 块，但是在面对大量顺序访问工作会暴露出和 LRU 算法相同的问题：在确认某一个 cache 块可能会常驻缓存之前，其针对大量顺序访问的数据内容容易丢失进入时间过长的数据。故该算法适用的环境如下：

1. 程序中具有小部分可以驻留在 cache 块中很久的数据块，且两次访问之间的间隔不过长
2. 有多个程序共享某个库文件或动态链接库

## 算法复杂度分析

#### 空间复杂度

PLRU算法与LRU算法使用相同的空间，均对每一个数据块使用4B的空间储存相关数据。可以认为是O(1)的空间复杂度

#### 时间复杂度

PLRU 算法需要多次确认一个 cache 行所有路中属性值的最大值，在不优化的前提下需要 O(n)的时间，n是组相联的路数。如果可以在硬件上和算法上进行优化，可以达到O(log(n))的时间复杂度。

## 算法测试结果展示

在提供的三个数据集上，PLRU 算法的测试结果如下（具体测试结果可见文件r1.txt,r2.txt与r3.txt）：

### 堡垒之夜

|  |  |
| --- | --- |
| 算法（排名前三） | 命中率 |
| RAND | 9.836136% |
| LRU | 9.764601% |
| FRU（3） | 11.137485% |
| SRRIP 3 | 10.544985% |
| SRRIP\_FP | 10.081127% |
| BRRIP（2） | 14.492665% |
| DRRIP | 10.660773% |
| PLRU（1） | 14.555045% |

### 和平精英

|  |  |
| --- | --- |
| 算法（排名前三） | 命中率 |
| RAND | 14.843126% |
| LRU（3） | 16.059805% |
| FRU | 11.233655% |
| SRRIP 3（2） | 16.098861% |
| SRRIP\_FP（1） | 16.212840% |
| BRRIP | 14.692341% |
| DRRIP | 16.009839% |
| PLRU | 15.535191% |

### 王者荣耀

|  |  |
| --- | --- |
| 算法（排名前三） | 命中率 |
| RAND | 16.272644% |
| LRU | 18.489758% |
| FRU（3） | 16.262376% |
| SRRIP 3 | 18.814370% |
| SRRIP\_FP | 18.873923% |
| BRRIP（2） | 19.585600% |
| DRRIP | 18.833313% |
| PLRU（1） | 19.558163% |

## 参考文献

[1] O’NEIL E J, O’NEIL P E, WEIKUM G. The LRU-K page replacement algorithm for database disk buffering [J]. SIGMOD Rec, 1993, 22(2): 297–306.

[2] JOHNSON T, SHASHA D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm; proceedings of the VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, F, 1994 [C].

[3] SONG J, ZHANG X. LIRS:an efficient low inter-reference recency set replacement policy to improve buffer cache performance [J]. 2002, 30(1): 31-42.

[4] QURESHI M K, JALEEL A, PATT Y N, et al. Adaptive insertion policies for high performance caching [J]. Acm Sigarch Computer Architecture News, 35(2): 381.

[5] JALEEL A, THEOBALD K, JR S, et al. High performance cache replacement using re-reference interval prediction (RRIP) [M]. 2010.