Elastic Queue: A Universal SSD Lifetime Extension Plug-in for Cache Replacement Algorithms

**1. Introduction**

【基本要重写，篇幅会较长】

1. SSD Cache重要性；②使用各种Cache算法的必要性）

P1. SSD Cache的重要性，已经在使用。历史上Cache算法有很多，很难说哪个算法一定最好，不同应用情景下，表现优异的算法很可能不同。给出一个我们的实验结果加以证实。如果经过测试知道某种算法在某个环境下表现特别出色，就可以设为这种策略来提高性能。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | LRU | LFU | LIRS | ARC |
| 云数据库 | 56.6% | **59.5%** | 58.4% | 58.4% |
| 随机文件访问 | 1.9% | 4.6% | 18.7% | **20.3%** |
| 网络文件系统 | 13.4% | **18.8%** | 15.4% | 15.8% |
| VOD | 9.24% | 4.0% | **17.3%** | 14.4% |

注：5%的Cache空间

（传统算法的不足；现有改进算法的不足；提出解决方案）

P2. 但是SSD Cache上面临耐久性问题，传统的Cache无法解决，因此有一些改进的Cache算法提出[引用SieveStore, LARC, L2ARC, WEC, ETD-Cache等]，但是已有算法选择缓存对象的方式确定(see Section 2.X和表XX)，不能充分利用各种Cache算法的优势，在各种应用环境中选择最合适的算法。所以需要一种新的方法，既能够发挥各种Cache策略的优势，又在其基础上显著减少SSD写入，延长SSD寿命。

(解释为什么新方法可以配合所有算法，并且可以解决寿命问题)

P3. 缓存算法都是用某种规则形成一个数据的排序（按照evict的优先级从低到高），如下图所示，其中实线框表示SSD缓存空间，虚线框表示假设将未缓存的其他所有数据也按照缓存算法进行排序。那么理想情况是缓存数据不断命中，始终停留在实线框中(如图a所示)。然而在实际应用中，由于缓存空间有限和好数据的热度不稳定这两个原因，好数据不加保护的话会不断通过边界，如b所示，造成很多写入量；通过动态调整边界的位置和将它们pin住，可以减少数据在边界来往的次数，减少额外的写入。从图c可以看出，采用这样的方式之后，避免了白块在边界的进出，达到了a的效果。



（说明新方法是可以起作用的）

P4. 我们的一个统计证实了在实际的应用中，情况b的发生频率还是很高的。以LRU为例，被缓存算法evict的数据中，有相当一部分是热门数据，只是暂时冷，就被踢出了。这就导致大量热门数据不能一直停留在缓存空间中，而是不断进出边界

//将原来3.2的内容和图3拿来，不谈no-hit rate了。（原表1省略，只在正文中用一句话提一下各个Trace代表什么应用）

（总结）

P5. 所以，已经缓存算法虽然多样，但是在SSD写入量过多的这个问题上，他们的问题是一样的，都是对这样的好数据没有保护好，让其evict，后面又会由于表现优异再次进入SSD Cache，循环往复，产生大量没必要的写入量。因此我们提出算法的思路就是Elastic Queue，在其他算法排序队列的基础上，判别好坏数据，将好数据Pin住，为好数据弹性增加队列长度，即使进入虚线区域也不evict。实验结果blablabla。

P6. 下文组织

**2. Related Work**

2.1 Write Endurance of SSDs

不用变

2.2 Cache Replacement Algorithms

P1 不变

P2 加入我们自己的工作引用，WEC和ETD-Cache

增加P3，说明以上经典算法，以及面向SSD的改进方法都是用固定的策略选择SSD缓存对象，不能很好适应不同应用特征。经典算法（LRU等）是倾向于recency，或frequency，或二者的混合，这里就不总结了。主要总结一下面向SSD的改进算法选择缓存对象的固定策略：

|  |  |
| --- | --- |
| 算法 | 选择缓存对象的固定策略 |
| SieveStore | Frequency，访问数达到阈值作为准入条件 |
| LARC | 两个LRU队列（再确认一下） |
| L2ARC | 内存队列ARC算法，SSD队列FIFO |
| WEC | LRU队列 |
| ETD-Cache | 两个LRU队列，加采样 |

**3. Elastic Queue**

【基本要重写，有些素材可以使用，或改动后使用】

**3.1 Principle of Elastic Queue**

尽管cache算法有各种各样的形式，但从整体的角度看，都可以看做是将数据按照evict优先级排列在队列中，然后淘汰高优先级数据，补充新数据的过程。所以不论算法的实现方式如何，都可以用一个缓存队列来描述，而不同算法之间的核心区别在于如何给数据划分优先级。

从之前的分析中可以看到，在如此多的应用场景中，针对某一个或某一类应用，一般都可以找到一种缓存算法，可以比其他算法更准确地鉴别出其中的热门数据。但是由于缓存空间有限，一旦这些数据离开了缓存队列，就会触发缓存替换，增加SSD写入。虽然这些数据离开了缓存空间，但我们仍然可以将它们看作延伸队列的一部分。如果一个数据被驱逐出了缓存空间，但在延伸队列中多次命中，不久后它会再次进入缓存空间。如此一来，这一次替换就带来了额外的写入，这在内存替换中并不是太严重的问题，但是在SSD这样的不耐久设备上就会白白损耗宝贵的寿命。

基于这个考虑，EQ的思想是将好数据保护起来，不让其由于短暂的冷却而被evict。

**3.2 Architecture of Elastic Queue**

在系统中有两类数据：General block, Pinned block(always on SSD).

PB是EQ由Pinning Block module负责将Cache size内的数据块pin住（如果考察其为好数据）；Pin的状态不会一直保持，由Unpinning Block module负责判断pinned block何时不好了，应该退出，变为unpin状态。

General blocks：除了Pinned block之外，其他的block也可能被原始的算法认为是值得缓存的数据，但由于EQ的存在，SSD空间被PB占据了，所以只记录GB的元数据。当有Pinned block退出时，这些GB将会被Pinning module考察和挑选成为Pinned block，真正进入SSD并停留一段时间。而如果未能成为Pinned block，而且超过了cache size，就不再被Pinning module考虑，而是直接evict，此时的evict本质上只是删除其在SSD queue中的元数据。

Elastic Queue就是在原有的优先级队列上延长了队列长度，容纳原本只能直接被Evict而现在被Pin住的数据。EQ的长度是弹性变化的，两种极端情况：当原有的优先级队列都是Pinned block时，EQ的长度等于原有优先级队列长度；当所有Pinned block都在延长队列中时，原有队列只有GB的元数据，此时队列长度为2倍。

// 介绍EQ和上述完全队列的对应关系：每次有数据退出，会加1，这样能看出来每个pinned block在上述虚拟队列中的位置。

只有cache size内才可能有general block，cache size外只可能有pinned block。

我们会统计每个数据块（包括general block和pinned block）的命中数，包括在cache size内的，和cache size外的，用于判定其热度。

// 介绍General Block和Pinned Block的处理规则区别

Pinned block真正存在于SSD中，general block只是元数据。

Pinned Block：由Pinning Block module负责将Cache size内的数据块pin住（如果考察其为好数据）；由Unpinning Block module负责判断pinned block的工作空间是否应该被回收，变为unpin状态。

General Block：过了cache size的线就evict（实际只有元数据）。



给出一个具体的例子，将原Fig. 5的图改造一下，符合EQ就行。

3.2部分要强调一下EQ名字的来历，我们把原来的优先级队列的长度可能延伸出去，但是长度是弹性变化的，可长可短，最长是2倍的元队列长度，最短是1倍。

但核心问题在于（1）如何判定哪些数据时值得保护的数据；以及（2）应该为这些数据增加多少工作空间让其发挥作用。

为了回答这两个问题，我们将从两个角度考察缓存数据——数据在缓存空间中的优先级变化规律，和数据的访问类型。

**3.3 缓存空间中的优先级变化规律**

每种应用环境下表现突出的算法，其选择数据、缓存数据的方式在这一环境下是比较有效的。因此，通过考察原有cache算法对缓存块在cache size中的评价（即优先级），可以了解在这个时间点上，这个块的热门程度。而将这种观察在一段时间内间隔进行，可以看出cache中数据块的热度变化。

这种方法可以看作隔一段时间对Cache中的优先级队列拍一个快照，记录队列中数据块当前的优先级。当需要pin住一个block时，可以从一段时间内采集的快照了解原有cache算法对这些数据块的评价，然后分析出哪些数据是在接下来的一段时间内值得停留在cache中的

举例：缓存队列中淘汰优先级不断增高的缓存块，说明原有cache算法认为它的热度在不断降低，将他留在cache中用处不大；而淘汰优先级不断降低，即热门程度不断攀升的块，说明根据原有cache算法，它在近期内对提高性能有很大作用，这时就应该将它保护起来，避免因为暂时靠后而被退出。（多加一个稳定的类型或者别的）



3.3.1 快照实现

细节、数据结构

**3.3.2**触发Pinning block

记录当前pinned block数量，由此知道SSD Cache是否还有空间。如果有空间就会触发pinned block，从EQ首部开始找第一个general block pin住。

// 原4.5 Location Transition内容改一改。

3.3.2 Black List

简要介绍，unpinned block要记录一下，这里的数据不能pin，会skip。

Black list是一个长度有限的FIFO队列，时间长了有可能再次进入。

**3.4 额外工作空间分配**

缓存数据的访问周期是不稳定的，即使是好数据（WEC），也会有一些访问的间隔较大。所以，为了让挑选出的好数据发挥作用，免受噪声的干扰而被驱逐，需要扩展它们的工作空间。EQ的弹性可以使pinned block有更多的工作空间。

不同的数据块热度不同，应该区别对待。如果我们pin住了一个非常好的数据，那么就应该对实施更有效的保护。既然偶然的长访问间隔不可避免，给它分配更大的工作空间可以确保这个数据不被驱逐。而如果某一次pin的块不是非常热门，那么就不应该浪费太多的Cache资源，以防之后想引进更好的数据时没有空间。

由于pinned block在cache中停留的时间长，仅仅从少量的快照不能推断出它的长期热度。我们采用了一种开销更低的方法，通过统计每个数据块在cache之外的虚拟队列中各个段的命中数，反映出它的长期性质。

3.4.1 数据分类

举例，直方图。

数据结构。

3.4.2 UnPinning Block的逻辑介绍.

**4. Evaluation**

先保留原内容，具体措辞上修改。

**5. Conclusion**

最后修改。