

**高级计算机体系结构**

**课程报告**

题目: SOSP’23论文*FIFO queues are all you need for*

*cache eviction*的复现与改进

院系: 系统科学与工程学院

专业: 计算机科学与技术

学生姓名: 杨珊

学 号: 23216635

时间： 2023年 12 月 05 日

目录

[1 摘要： 2](#_Toc152696882)

[1.1 关于原文 2](#_Toc152696883)

[1.2 关于复现与尝试改进 2](#_Toc152696884)

[2 简介： 3](#_Toc152696885)

[3 研究背景： 5](#_Toc152696886)

[3.1 如何衡量Cache替换算法的优劣 5](#_Toc152696887)

[3.2 常见的基于LRU的Cache替换算法 6](#_Toc152696888)

[4 提出本文算法的动机 7](#_Toc152696889)

[4.1 一次命中奇迹的比例比我们想象得要高 7](#_Toc152696890)

[4.2 快速降级的需求 9](#_Toc152696891)

[5 设计与实现 11](#_Toc152696892)

[5.1 S3-FIFO 设计 11](#_Toc152696893)

[5.2 实现 12](#_Toc152696894)

[6 开销分析 13](#_Toc152696895)

[7 评估 14](#_Toc152696896)

[8 复现与尝试的改进 15](#_Toc152696897)

[8.1 复现 15](#_Toc152696898)

[8.1.1 复现流程 15](#_Toc152696899)

[8.1.2 指标复现 17](#_Toc152696900)

[8.2 尝试改进 19](#_Toc152696901)

[9 总结 21](#_Toc152696902)

# **摘要**：

## 关于原文

来自SOSP‘23（ACM Symposium on Operating Systems Principles 2023）的论文 *FIFO queues are all you need for cache eviction* 提出了一种全新，高效，可扩展，对闪存友好的Cache替换算法**S3-FIFO**，作者Juncheng Yang等人洞察到，不论是模拟的Cache访问序列还是实际生产数据中的Cache访问序列，在Cache容量比较小的情况下，很多数据只进入Cache一次就被淘汰走了，因此如果用LRU等复杂算法开销是很大的，因此设计了一个使用三个FIFO队列的Cache替换算法，并在实际生产数据的Cache访问序列中表现优秀。

FIFO（First In First Out）作为一种Cache 替换算法， 具有很多吸引人的特性：算法简单，速度快，可扩展性强，对闪存的友好性。而FIFO被认为最突出的缺点是其未命中率很高。在本文中，作者展示了一种基于FIFO的简单、扩展性强的Cache替换算法，其中包括了三个静态队列：Small FIFO，Main FIFO 和 Ghost FIFO，故命名为S3-FIFO。作者在14个数据集上的6594个Cache访问序列对各种Cache替换算法进行评估，发现S3-FIFO在各种访问序列的命中率低于目前最先进的算法。S3-FIFO的示例图如图表 1 S3-FIFO的设计图 所示。

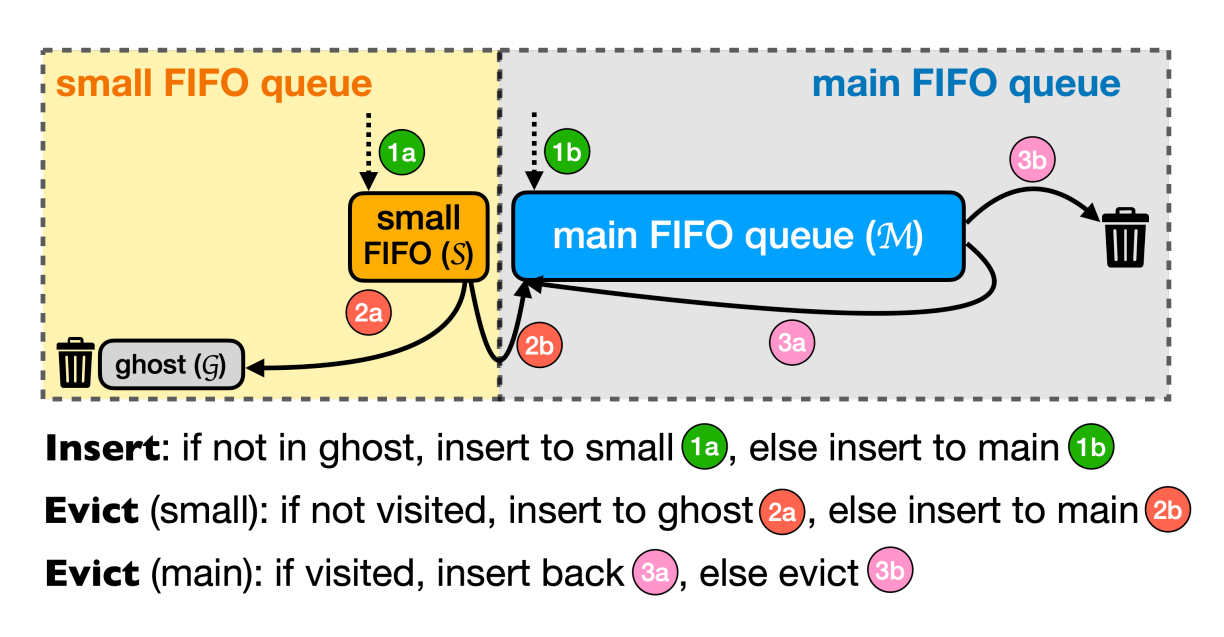
此外，S3-FIFO的效率非常稳健，它在14个数据集中有10个数据集上的平均未命中率均为最低。FIFOI队列使得S3-FIFO能够以比优化过的LRU（Least Recently Used）算法高6倍的吞吐量在16个线程上运行。

作者之所以洞察到可以用FIFO来设计Cache替换算法，是因为在倾斜的工作负载中，大多数对象在短时间窗口内只会被访问一次，因此很有必要将他们从Cache中替换出去（也称为Quick Demotion，快速降级）。S3-FIFO的关键在于一个其包含Small FIFO队列，它可以阻止大部分对象进入Main FIFO，从而提高了可靠的快速降级和高降级精度。

## 关于复现与尝试改进

本报告使用CacheSim模拟器复现了论文中提出的S3-FIFO算法，并且基于复现的算法，画出了论文中的部分图片，展示了论文中算法的可行性和高效性。

此外，本报告使用rust模拟实现S3-FIFO后，尝试改进了原先的算法，并对改进的思想进行了一些论证。改进的代码于仓库的rust-implementation&improve文件夹中。



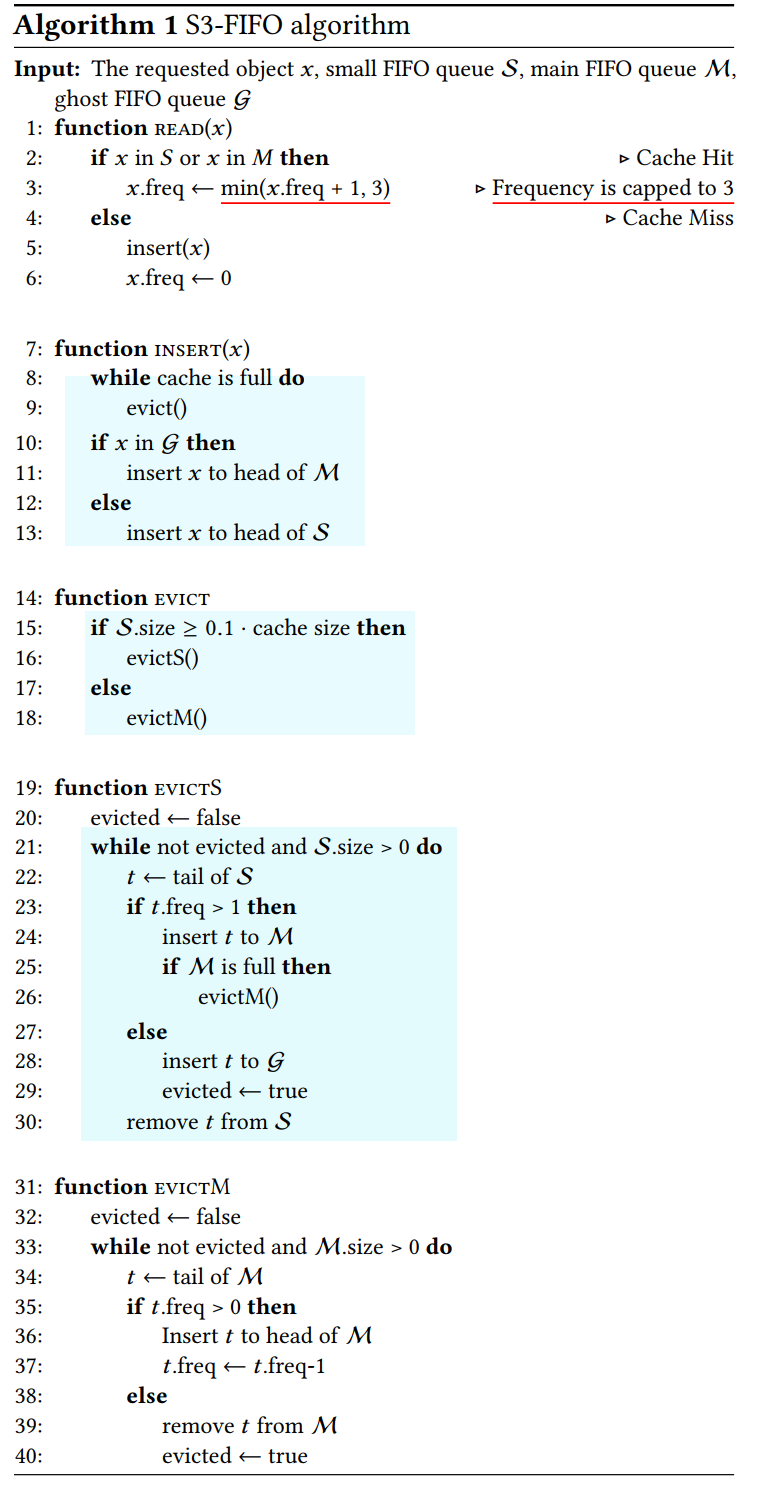
图表 1 S3-FIFO的设计图

# 简介：

随着Cache与后端存储系统之间的延迟缩小，以及CPU的核心数量迅速增加，Cache的吞吐量和可扩展性变得至关重要。在过去的几年里，越来越多的文献对此进行了研究，解决方案通常是通过使用简单的FIFO队列Cache替换算法来权衡效率以换取吞吐量和可扩展性。例如，MemC3 [57]、Tricache [60] 使用CLOCK算法，而Segcache [158] 使用FIFO合并。与LRU相比，FIFO更简单、更可扩展，但效率较低。本文作者仅使用FIFO队列构建一种简单的、可扩展性强的Cache替换算法。在Cache的工作负载中，对象的热门程度通常是不平衡的，并且遵行幂律分布（比如Zipf分布）。作者洞察到对于任何Zipf访问序列，只出现一次的对象（被称为One Hit Wonder，一次命中奇迹）在子序列中的比例要比整个访问序列中高得多。尽管大多数对象在整个访问序列中有被多次访问，但是因为大小为C的Cache在Cache已满时只能观察到C个对象的短序列，所以大多数对象在被淘汰时将只会被访问一次（插入Cache到被淘汰时没有其他访问请求）。

作者在6594个生产环境的访问序列中确认了这一现象。考虑整个访问序列时，所有访问序列的中位数一次命中奇迹的比例为26%；然而当关注每个访问序列的10%的子序列时，中位数一次命中奇迹的比例激增至72%。

作者利用了上述的这种特性设计了S3-FIFO。具体的算法如图图表 2 S3-FIFO算法流程图所示。



图表 2 S3-FIFO算法流程图

# 研究背景：

## 如何衡量Cache替换算法的优劣

Cache的核心是Cache替换算法，它决定了在有限的Cache空间要存储哪些对象。衡量一个Cache替换算法好与不好通常看以下的指标：

1. 效率：一个更高效（有时被称为更“有效”）的Cache替换算法会在Cache中保留更多有用的对象，并提供更低的未命中率，未命中率衡量了必须从后端存储系统（比如磁盘）获取的请求的比例。虽然请求未命中率是最常见的效率指标，但一些旨在减少带宽使用的Cache部署（例如代理Cache）还评估字节未命中率：需要从源获取的字节数的比例。
2. 吞吐量：Cache的吞吐量衡量它每秒可以处理的请求数（QPS，Requests it can serve Per Second）。具有更高的吞吐量可以减少需要为工作负载提供服务的CPU核心数量。
3. 可扩展性：现代CPU具有大量的核心。例如，AMD EPYC 9654P拥有192个核心。Cache的可扩展性衡量了它的吞吐量如何随着CPU核心数量的增加而增加。理想情况下，Cache的吞吐量应该与CPU核心数量成线性关系。然而，在许多Cache替换算法中，读操作需要在锁定Cache的情况下进行数据更新。因此，它们无法充分利用现代CPU的计算能力。
4. 闪存写入：虽然DRAM是Cache的最常用存储介质，但今天许多系统也使用闪存作为Cache，因为它具有更高的密度、更低的价格和更低的功耗。当将闪存用于制作Cache时，闪存寿命成为一个关键指标，因为闪存只支持有限数量的写入。此外，闪存上的小随机写入会引起设备级别的写入放大，这不仅会减少闪存寿命，还会增加读取和写入尾延迟。为了实现更可管理的闪存寿命，大多数生产级别的闪存Cache系统（例如Apache Trafficserver、Memcached Extstore、Cachelib large object Cache和Google Colossus Flash Cache）使用FIFO或FIFO-reinsertion。除了闪存Cache替换算法，许多系统还采用入场算法（admission algorithm），例如布隆过滤器（bloom filter）或基于机器学习的算法，来选择要写入闪存的“好”数据。
5. 简单性和通用性：Cache替换算法的复杂性和通用性是对其采用起着关键作用的两个因素。尽管复杂性通常与吞吐量和可扩展性成反比，但简单的设计可以提供超出性能指标改进之外的好处，例如减少错误和降低维护开销。Linux内核开发人员曾表示，“预测哪些页面将在不久的将来被访问是一项棘手的任务。内核不仅经常犯错，而且浪费了大量CPU时间来做出不正确的选择”。通用性也是至关重要的，因为相同的数据结构和Cache替换算法可以用于不同类型的Cache，有助于减少开发和维护开销。在Meta[24]的以前工作中也可以找到类似的论点。

## 常见的基于LRU的Cache替换算法

LRU (Least Recently Used)是一种常见的Cache淘汰算法，它基于时间局部性的观点，即最近访问的数据更有可能被再次访问。因此，在DRAM（动态随机存取存储器）Cache中，LRU比FIFO更高效，并被广泛使用。此外，旨在提高效率的先进淘汰算法大多基于LRU构建。例如，ARC [100]、SLRU [80]、2Q [79]、EELRU [124] 和 CACHEUS [119] 都使用一个或多个LRU队列来排序Cache中的对象。

尽管LRU算法和基于LRU的改进算法是很高效的，但它们存在三个问题。

首先，LRU通常使用双向链表实现，每个对象需要两个指针，当对象很小时，这会导致较大的开销。因此，Twitter和Meta针对包含小对象的工作负载设计了专用的紧凑型Cache[24, 48, 158]。

其次，LRU在每次Cache命中时将对象提升到Cache队列的头部（称为提升），这会执行至少六次受锁保护的随机内存访问，严重限制了Cache的可扩展性[60, 112]。例如，RocksDB的开发人员承认LRUCache是可扩展性瓶颈[50]。因此，于2022年已经实施了使用CLOCK [45]淘汰的新Cache以解决这个问题[117]。

最后，LRU对闪存不友好。LRU中的对象淘汰顺序与插入顺序不同，这会导致在闪存上进行比较多的随机写入，并降低闪存寿命。

# 提出本文算法的动机

尽管过去几十年的淘汰算法研究都集中在LRU上，但我们认为现代淘汰算法应该使用FIFO队列而不是LRU队列进行设计。FIFO可以使用环形缓冲区实现，无需每个对象的指针元数据，并且它不会在每次Cache命中时提升对象的优先级，因此消除了可扩展性瓶颈。然而，普通的FIFO算法在效率方面落后于LRU算法和最先进的淘汰算法。

FIFO的主要局限性是无法保留频繁访问的对象，因此最直接的改进是将这些对象重新插入。FIFO-Reinsertion是一种FIFO的改进算法，用于跟踪对象访问并在淘汰过程中重新插入已访问的对象。与LRU相比，FIFO-Reinsertion在Cache命中时产生较低的开销，只需在首次对对象的请求中执行无操作或仅执行原子设置操作。然而，仅仅重新插入是不够的，FIFO-Reinsertion在效率方面仍然落后于最先进的淘汰算法。

我们的观点是，Cache会经历比常见的完整访问序列分析所建议的更多的一次命中奇迹（即在插入后没有访问的对象），突出了迅速删除大多数新对象的重要性。具体来说，在6594个生产环境的访问序列中确认了这一现象。考虑整个访问序列时，所有访问序列的中位数一次命中奇迹的比例为26%；然而当关注每个访问序列的10%的子序列时，中位数一次命中奇迹的比例激增至72%。

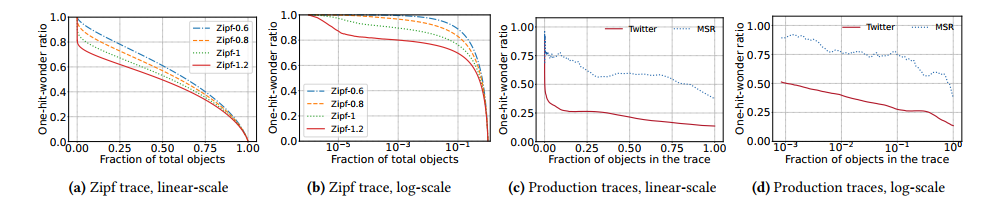
## 一次命中奇迹的比例比我们想象得要高

"一次命中奇迹比例"衡量了在跟踪中仅请求一次的对象所占的比例。由于大量的一次命中奇迹比例，它在内容传送网络（Content Delivery Networks，CDNs）中通常被广泛使用[19, 96]。

尽管一次命中奇迹比例在不同类型的Cache工作负载之间不同，但我们发现较短的请求序列（包含较少的唯一对象）通常具有较高的一次命中奇迹比例。在随后的分析中，我们使用唯一对象的数量来衡量序列的长度。

假设请求序列的对象被经常访问的程度遵循Zipf分布，最不被经常访问的对象只有一个请求，并且总共有𝑀个唯一对象。然后，完整序列的一次命中奇迹比例为1/𝑀。对于以一次命中奇迹结尾的任何子序列，如果子序列包含𝐶个唯一对象，则随着对象数量为序列长度𝑥（以对象数量衡量）的增加，期望的一次命中奇迹比例F(𝑥 = 𝐶)单调下降。

直观的解释是大多数对象不受欢迎（在大小为𝐶的Cache的Zipf分布中排名高于𝐶+1），并且其期望请求数量在0和1之间。如果它们出现在子序列中，它们很可能不会在子序列内再次被请求。这种设置可以看作是优惠券收藏问题的变体，其中我们总共有𝑀张唯一的优惠券，收集优惠券𝑖的概率遵循Zipf分布。我们想知道当我们有𝐶张唯一优惠券时，我们只收集了一次优惠券的数量。

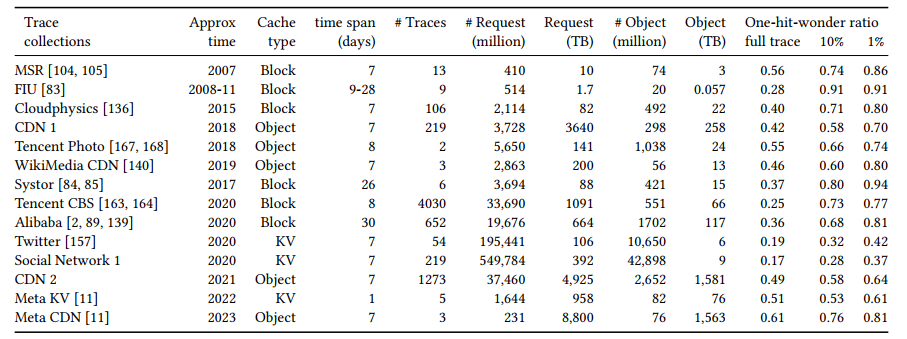


图表 3 模拟访问序列和实际生产访问序列的一次命中奇迹比例

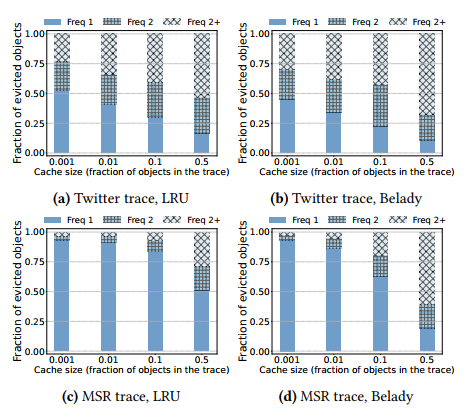
我们使用蒙特卡洛模拟来找出F(𝑥)随序列长度𝑥（以对象数量衡量）的变化。首先，我们根据独立参考模型[38]生成不同偏斜度𝛼的Zipf请求跟踪，然后取随机子序列并测量一次命中奇迹比例。我们重复100次并报告均值。结果绘制在图3a和图3b中，为了清晰起见，我们显示线性和对数刻度的X轴。一次命中奇迹比例随着序列长度的增加而减少。在不同曲线之间，更偏斜的工作负载在相同的序列长度下展现出较低的一次命中奇迹比例，因为不受欢迎的对象在更偏斜的工作负载中出现的概率较低。

我们还在生产环境的访问序列上进行了相同的测量。图3c和图3d显示了一个块跟踪（MSR hm\_0）和Twitter的Web跟踪（cluster 52）。这些曲线乍看之下与Zipf曲线不同。这是因为生产环境的访问序列不足以捕获后端系统中的所有对象，也不可能知道可以请求的总对象数。因此，X轴显示了跟踪中对象的比例。因此，生产曲线仅捕获了合成曲线的左侧区域，我们观察到它们与合成曲线相匹配。例如，比较图3a和图3c，我们可以看到这两幅图中的曲线在降低速度之前都有急剧下降。此外，已知Twitter跟踪更偏斜[157]，并且显示出比MSR跟踪更大的下降，与Zipf跟踪的观察相匹配。与全跟踪的一次命中奇迹比例相比（Twitter为13%，MSR为38%），包含10%对象的随机子序列在Twitter跟踪上显示一次命中奇迹比例为26%，在MSR跟踪上为75%。当进一步缩短序列长度时，这种增加更为显著。

表格 1不同数据集的评估



我们进一步使用了6594个生产环境中的访问序列进行了评估（详见表1）。图4显示了不同序列长度的所有跟踪的一次命中奇迹比例。与具有中位数一次命中奇迹比例为26%的全跟踪相比，包含跟踪中50%对象的序列显示了中位数一次命中奇迹比例为38%。此外，包含10%和1%对象的序列分别具有72%和78%的一次命中奇迹比例。



图表 4 不同生产访问序列在不同Cache长度下的一次命中奇迹

由于Cache大小始终远小于跟踪印记（跟踪中的对象数量），淘汰在遇到短序列请求后开始。这一观察表明，如果将Cache大小设置为跟踪印记的10%或1%，则大约72%和78%的对象在淘汰前将不会被再次使用。

我们进一步通过Cache模拟来证实这一观察。图4显示了淘汰时对象频率的分布。我们的跟踪分析（图2d）显示，对于跟踪长度为10%的序列，Twitter跟踪具有26%的一次命中奇迹比例。模拟显示类似的结果：在跟踪印记的10%Cache大小下，由LRU和Belady淘汰的对象中有26%和24%在插入后不再请求。同样，MSR跟踪显示出更高的一次命中奇迹比例，为10%跟踪长度的序列（图2d），图4显示在LRU和Belady淘汰的对象中有82%和68%没有再次使用。这表明这些一次命中奇迹通常是良好的淘汰候选对象，也许不需要高度复杂的淘汰算法。

## 快速降级的需求

根据观察，Cache应该过滤掉这些一次命中奇迹，因为它们占用空间但没有提供任何好处。在CDN中，通常采用布隆过滤器来拒绝一次命中奇迹进入Cache[96, 141]。然而，布隆过滤器拒绝对象得太快，缺乏精确性，因为它拒绝所有以前未见过的对象。这导致所有对象的第二次请求都成为Cache未命中，通常导致效率一般。

过滤掉一次命中奇迹与设计抗扫描Cache淘汰算法有些相似，因为在扫描期间请求的对象通常是一次命中奇迹。研究人员已经开发了各种算法，用于存储工作负载，可以避免由扫描请求引起的Cache污染和抖动，例如ARC [100]、LRU-K [110]、CACHEUS [119]和LHD [21]。然而，现有的算法无法保证一次命中奇迹在被移除之前停留在Cache中的最短和最长时间。我们发现这些算法有时淘汰得太快或太慢，而它们的复杂性使得难以理解它们的行为。

这引发了一个问题：我们是否可以简单地使用一个小的试用性FIFO队列，以保证一次命中奇迹在插入一定数量的对象后被移除？

# 设计与实现

如2.1所提到的，Cache淘汰算法需要在高效之外还要简单和可扩展。本节介绍了S3-FIFO，一个仅由静态FIFO队列组成的简单且可扩展的淘汰算法。

我们首先定义LRU队列和FIFO队列。LRU队列在Cache命中时通过将请求的对象提升到队列头部来更新对象顺序。FIFO队列在Cache命中时不更新排序，对象按照插入顺序被淘汰。然而，被淘汰的对象可以重新插入队列以保留热对象。如2.2所述，大多数淘汰算法都使用LRU队列构建，只有少数算法，例如FIFO-Reinsertion，使用FIFO队列，因为传统智慧认为LRU队列可以提供更低的未命中率。

## S3-FIFO 设计

S3-FIFO 使用了三个FIFO队列：一个小FIFO队列（S），一个主要FIFO队列（M）和一个幽灵FIFO队列（G）。我们选择 S 使用Cache空间的 10%，这是基于对 10 个跟踪进行实验的结果，发现 10% 的泛化效果很好。然后，M 使用 90% 的Cache空间。幽灵队列 G 存储与 M 相同数量的幽灵条目（无数据）。

Cache读取。S3-FIFO 使用两位来跟踪对象的访问状态 [155]，类似于带有频率上限为 3 的封顶计数器。S3-FIFO 中的Cache命中会原子性地将计数器增加 1。需要注意的是，对于热门对象的大多数请求不需要更新。

Cache写入。新对象如果不在 G 中则插入到 S 中。否则，它会被插入到 M 中。当 S 满时，尾部的对象会被移动到 M 中，如果它被访问了多次，或者移动到 G 中，如果没有被访问。在移动过程中，访问位会被清除。当 G 满时，它会按照 FIFO 的顺序淘汰对象。M 使用类似于 FIFO-Reinsertion 的算法，但使用两位来跟踪访问信息。已经被访问过至少一次的对象会被重新插入，其中一位被设置为 0（类似于将频率减 1）。我们在图 5 中演示了算法，伪代码见算法 1。

在3.1中，我们确定的一个重要模式是Cache由于有限的Cache空间而经历的大量一次性请求（one-hit-wonders）比例。小的FIFO队列 S 可以快速淘汰这些一次性请求，以防止它们长时间占据Cache。这允许 S3-FIFO 为更有价值的对象节省宝贵的Cache空间。除了由于倾斜的Cache工作负载中不受欢迎的对象而导致的一次性请求之外，许多块Cache工作负载还具有扫描和循环访问模式。与一次性请求一样，扫描期间访问的块会被迅速删除，以避免Cache污染和抖动。然而，在扫描中混合的未参与扫描的块也会在此过程中移动到 G。尽管如此，当这些“好”的块在不久的将来再次请求时，它们将被插入到 M 中并保留更长的时间。

## 实现

FIFO队列可以使用链表或环形缓冲区来实现。基于链表的实现可以更容易地添加到现有的基于LRU的Cache中。然而，它有三个缺点。首先，它使用每个对象两个指针。在具有小对象的工作负载上，这会造成巨大的存储开销。其次，遍历队列需要随机内存访问。第三，链表中的驱逐和插入需要昂贵的原子操作：比较和设置，这降低了可扩展性。

相比之下，基于环形缓冲区的实现具有更低的开销和更高的可扩展性，但可能不兼容现有的基于LRU的Cache系统。使用环形缓冲区来实现S3-FIFO时，环形缓冲区维护FIFO顺序，每个插槽存储对象或指针。淘汰需要在环形缓冲区中提升尾部指针。尽管在存储开销更低且更具可扩展性方面更具优势，但在工作负载包含许多删除操作时，环形缓冲区的空间可能会被浪费，因为删除的对象的空间直到淘汰才能被重用。我们经验性地观察到，在许多工作负载中，删除操作通常会在插入后不久到来，这使得S3-FIFO比传统的基于环形缓冲区的算法（例如CLOCK）更适合。这是因为如果被删除的对象在小的FIFO队列中，那么被删除对象的空间可以更快地被重用。

虽然S3-FIFO具有三个逻辑FIFO队列，但也可以使用一个或两个FIFO队列来实现。因为从S中淘汰的对象可能会进入M，所以它们可以使用一个队列来实现，带有指向10%标记的指针。然而，将S和M合并会降低可扩展性，因为从队列中间删除对象需要锁定。

幽灵FIFO队列 G 可以作为索引结构的一部分来实现。例如，我们可以在基于桶的哈希表中存储幽灵条目的对象指纹和插入时间 [33, 37, 93, 158]。指纹是对象ID的4字节哈希。插入时间是一个虚拟时间戳，统计迄今为止插入到 G 中的对象数量。让 𝑆G 表示幽灵队列的大小。如果当前时间是 𝑁（即迄今为止有 𝑁 个插入到 G 中），那么所有时间戳低于 𝑁 − 𝑆G 的条目都不再在 G 中。在对象被请求时或哈希冲突时，幽灵条目将从哈希表中删除。

# 开销分析

1. 计算

S3-FIFO在对对象的第一次和第二次请求时执行原子写入操作，无需锁定。在第二次请求之后没有其他操作。由于大多数请求都是对流行对象的请求（超过两次请求），因此S3-FIFO在Cache命中时执行可忽略不计的元数据更新。

Cache未命中需要从 S 或 M 中淘汰对象。从 S 中淘汰需要将尾对象插入到 M 或 G 中。而从 M 中淘汰可能涉及将尾对象重新插入到 M 中。但是，如果对象没有被访问，就不需要重新插入。因此，在实践中，重新插入的次数远小于Cache命中的次数。此外，删除尾对象并将对象插入队列头部可以使用原子操作实现无锁操作。

1. 存储

幽灵队列 G 存储的对象数量与主队列相同，但没有数据。假设平均对象大小为4 KB，对象ID使用4字节，那么 G 使用Cache大小的0.09%。每个Cache对象使用两位来跟踪访问，占用Cache大小的不到0.01%。此外，这两位通常可以附加在对象元数据中未使用的位上。如果使用环形缓冲区实现FIFO队列，S3-FIFO可以去除两个LRU指针，每个对象节省16字节，相当于Cache大小的0.4%。

通过使用Small FIFO队列来过滤对象，以实现超过最先进Cache替换算法的效率对于闪存Cache部署具有重要意义。如果Small FIFO队列位于DRAM中，而Main FIFO队列位于闪存中，那么从DRAM中淘汰的大多数对象不需要写入闪存。这既减少了闪存写入，又降低了未命中率。作者将这个FIFO过滤器（即Small FIFO队列）与Flashield [55]中的概率过滤器和基于机器学习模型的过滤器进行了比较。在两个开源内容传送网络（Content Delivery Networks，CDNs）访问序列上评估后，FIFO过滤器具有最低的未命中率和最少的闪存写入。此外，与需要大型DRAMCache（总Cache大小的10%）以跟踪对象访问信息以做出良好决策的ML模型不同，当DRAMCache仅占总Cache大小的0.1%时，Small FIFO过滤器仍然表现出色。

# 评估

作者对S3-FIFO算法进行了效率评估，使用了6594个生产追踪数据进行了测试。这些数据包括自2007年至2023年的不同类型Cache（键值、块和对象CDNCache）的请求和流量数据。评估主要集中在Cache性能的关键指标——未命中比率上。S3-FIFO算法与其他先进的淘汰算法进行了比较，这些算法不仅在历史上有意义，也在现有的生产系统中部署使用。模拟结果表明，在大多数评估的追踪数据中，S3-FIFO在较大的Cache大小下，常常表现优于其他算法，显著降低了未命中比率。

特别指出的是，S3-FIFO比TinyLFU更有效，尽管TinyLFU的性能不错，但它并不一致地在所有追踪中胜过FIFO，特别是在较小Cache大小下。文章指出TinyLFU中LRU窗口的大小过小，导致过早淘汰，并建议增大窗口大小可以提高效率。LIRS是另一个表现良好的算法，因其快速降级策略而效率高，但在某些情况下由于其实现比S3-FIFO复杂而表现不足。2Q算法与S3-FIFO设计相似，但由于其大的试用队列和缺乏重新插入LRU队列，效果不佳。文章还讨论了SLRU、ARC、CACHEUS、LeCaR、LHD和FIFO-Merge等其他算法，注意到虽然这些算法在某些条件下表现良好，但通常不如传统算法，特别是在块Cache工作负载中。

缺乏快速降级是效果不佳算法的一个共同主题，这对效率至关重要。例如，B-LRU采取了激进的策略拒绝一次性访问对象，但最终在大多数情况下比LRU的性能还差。文章识别了S3-FIFO表现不佳的对抗性工作负载，通常是对象只被访问两次且第二次请求落在小FIFO队列之外，导致Cache未命中。然而，这些条件很少见，通常可以通过自适应算法来缓解。

总结来说，S3-FIFO在广泛的数据集和Cache大小中表现出优越的性能和鲁棒性。在大Cache大小的14个数据集中，它在10个数据集上表现最佳，在小Cache大小的数据集中有7个。文章最后得出结论，S3-FIFO的简单性和其快速降级策略是其效率的关键，使其成为现代Cache淘汰挑战中一种强有力的解决方案

# 复现与尝试的改进

## 复现

### 复现流程

**操作系统：**

本报告使用的操作系统是Ubuntu22.04 LTS。

**CacheSim安装：**

作者使用开源的libCacheSim来模拟实现S3-FIFO算法。我们需要在操作系统中安装xgboost，lightbgm，zstd等依赖，然后再下载编译libCacheSim源码。

**数据集下载：**

需要从网站下载需要的数据集，里面包括了生产数据的访问序列：<https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/cacheDatasets/>

**数据处理：**

本报告使用zstd解压数据集后，使用原作者提供的脚本scripts/plot\_one\_hit\_zipf.py来画出Zipf分布访问序列的一次命中奇迹图标。

对于实际生产数据，我们需要下载实际生产数据后解压并通过scripts/plot\_one\_hit\_trace.py脚本画出实际生产数据时的一次命中奇迹的比例图。

**模拟运行：**

用bash脚本运行libCacheSim模拟不同Cache容量时使用LRU，S3-FIFO等算法在生产数据访问序列中的替换和命中情况。并且输出模拟运行数据至文件中。

使用作者提供的脚本scripts/plot\_throughput.py我们可以分析之前模拟输出的文件，并绘制图表展示不同算法的吞吐量。

# run LRU caches on the Twitter trace at cache size 0.001, 0.01, 0.1 and 0.5 of #obj in the trace

for cache\_size in 0.001 0.01 0.1 0.5; do

./libCacheSim/\_build/bin/cachesim twitter.oracleGeneral.bin.zst oracleGeneral LRU ${cache\_size} --ignore-obj-size 1 > twitter\_lru\_${cache\_size} &

done

# run Belady caches on the Twitter trace

for cache\_size in 0.001 0.01 0.1 0.5; do

./libCacheSim/\_build/bin/cachesim twitter.oracleGeneral.bin.zst oracleGeneral Belady ${cache\_size} --ignore-obj-size 1 > twitter\_belady\_${cache\_size} &

done

# use size 0 to choose cache size based on the trace footprint

# it uses cache sizes 0.001, 0.003, 0.01, 0.03, 0.1, 0.2, 0.4, 0.8 of the number of objects or bytes in the trace

./libCacheSim/\_build/bin/cachesim /path/to/data oracleGeneral algo 0 --ignore-obj-size 1

# an example

./libCacheSim/\_build/bin/cachesim msr.oracleGeneral.bin oracleGeneral lru,s3fifo 0 --ignore-obj-size 1

使用作者提供的脚本scripts/libCacheSim/plot\_demotion.py复现快速降级的图表

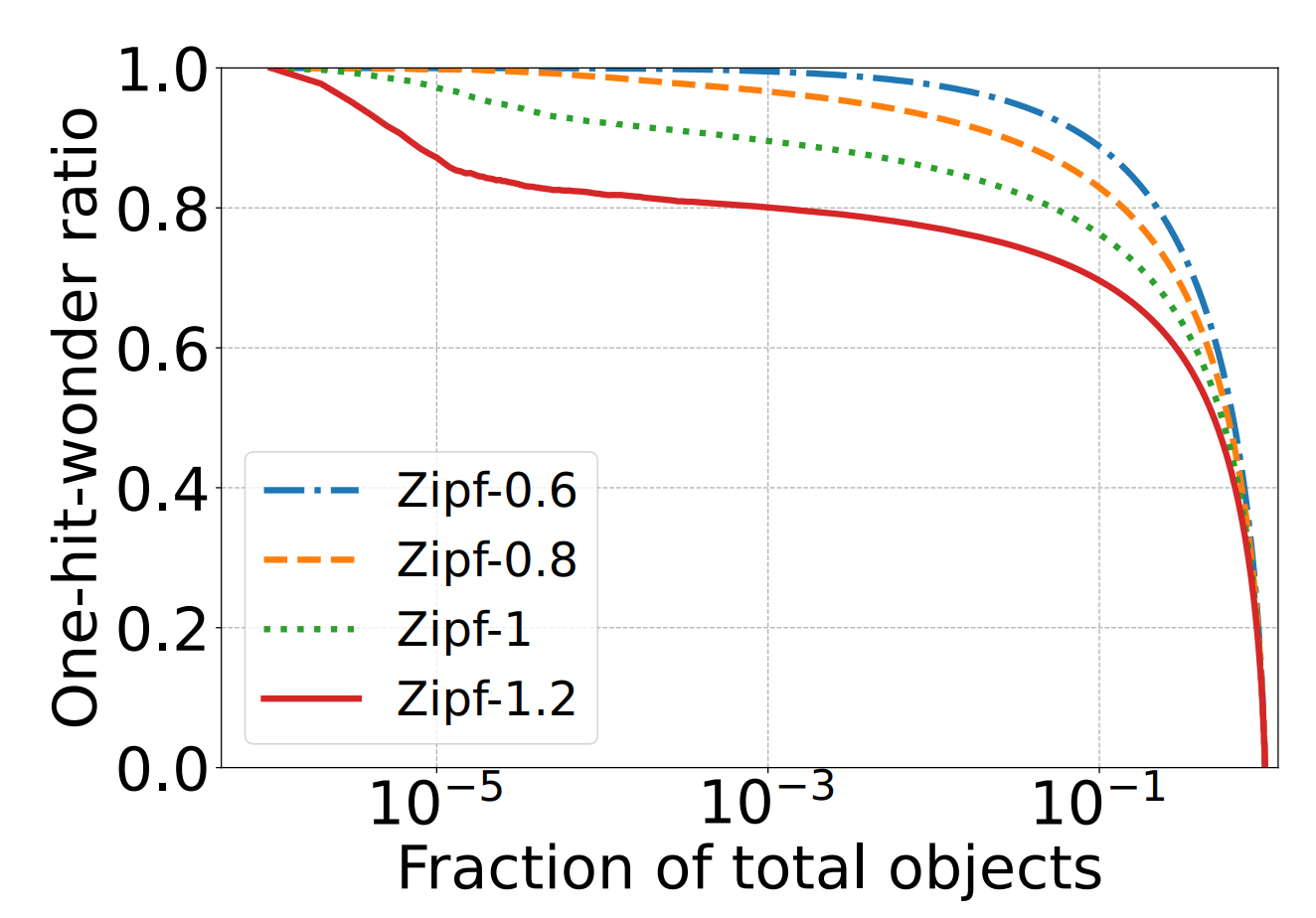
# MSR and large size

python3 scripts/libCacheSim/plot\_demotion.py plot --datapath result/demotion/demotion\_0.1 --dataname MSR

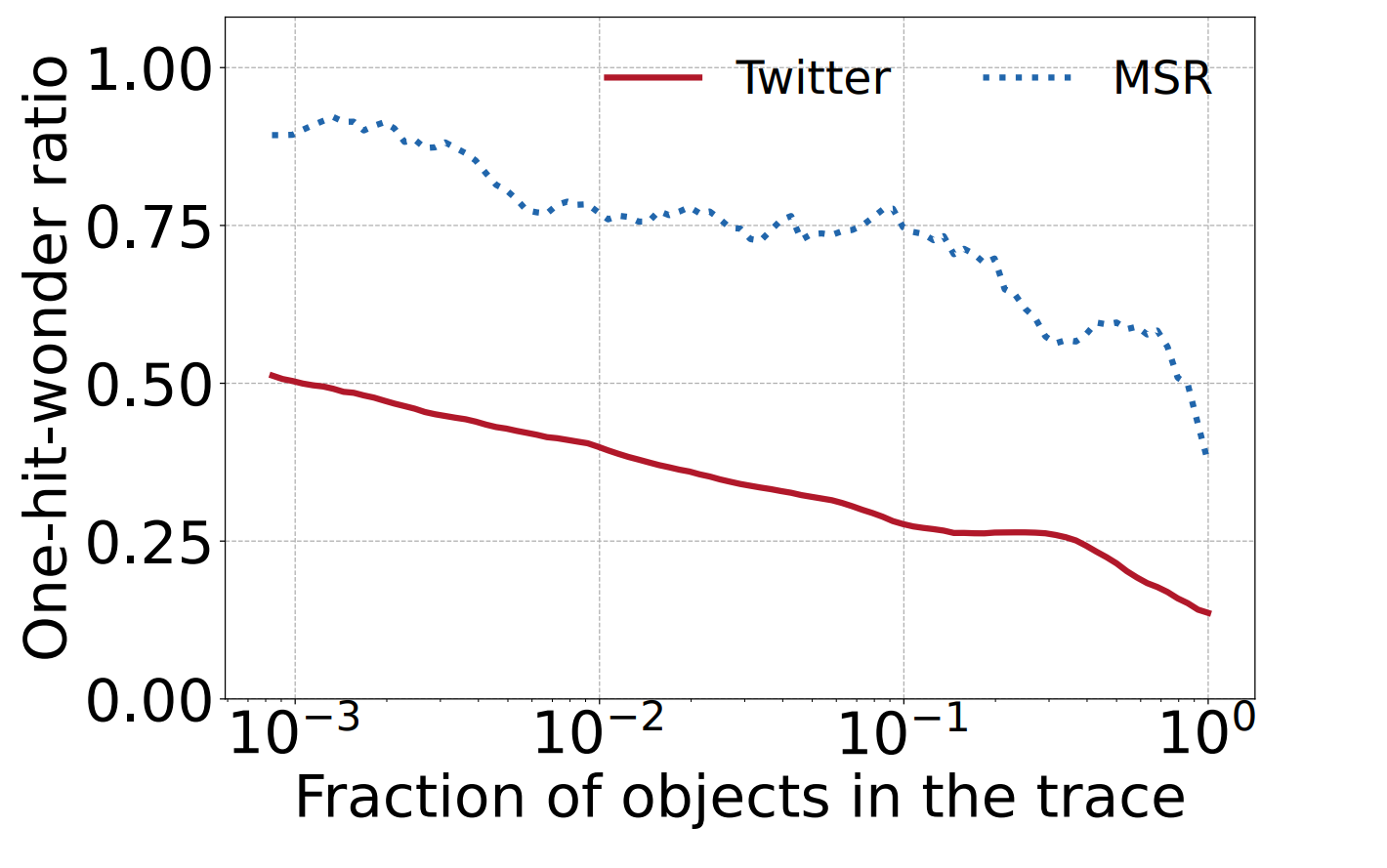
# Twitter and large size

python3 scripts/libCacheSim/plot\_demotion.py plot --datapath result/demotion/demotion\_0.1 --dataname twitter

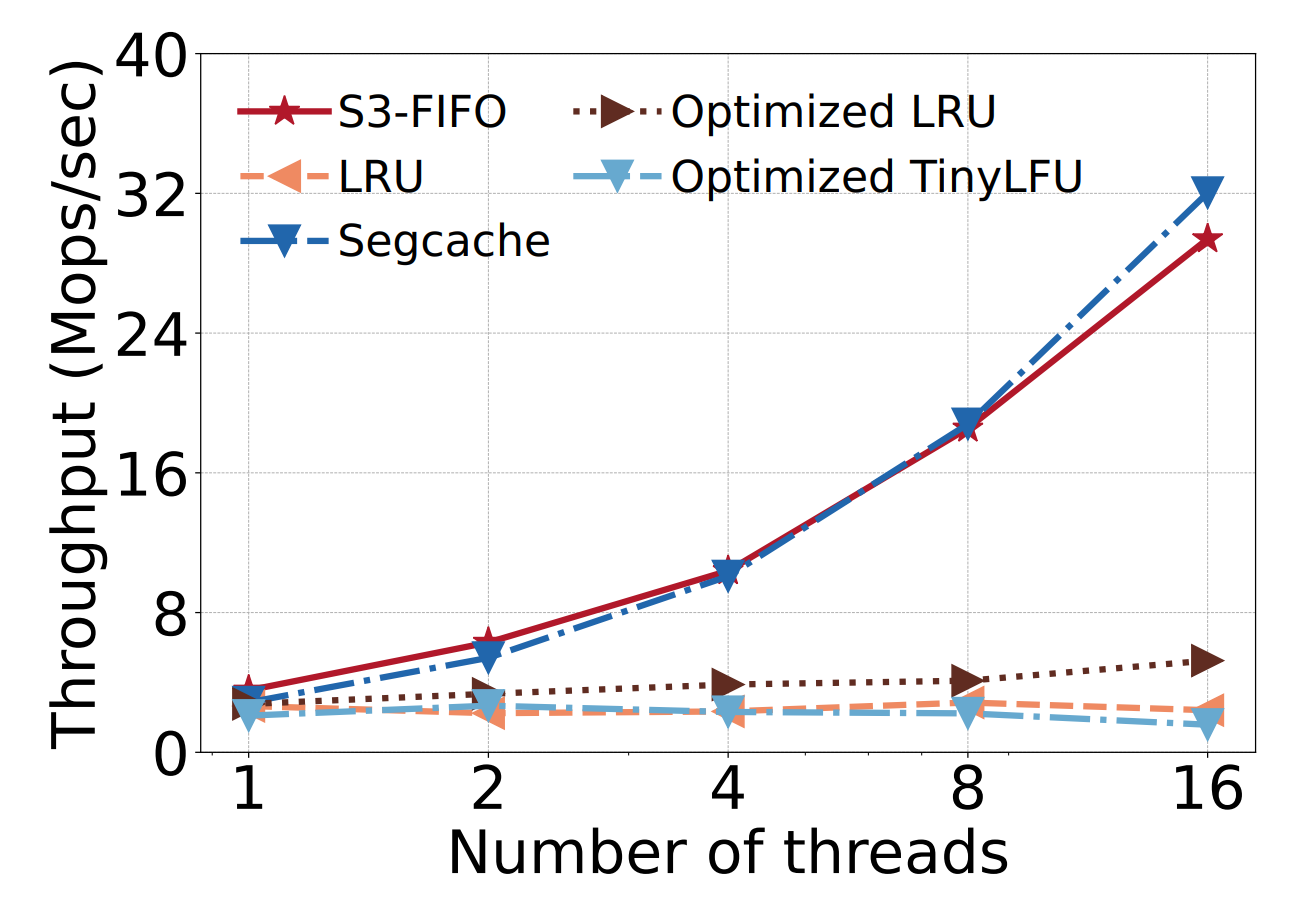
### 指标复现



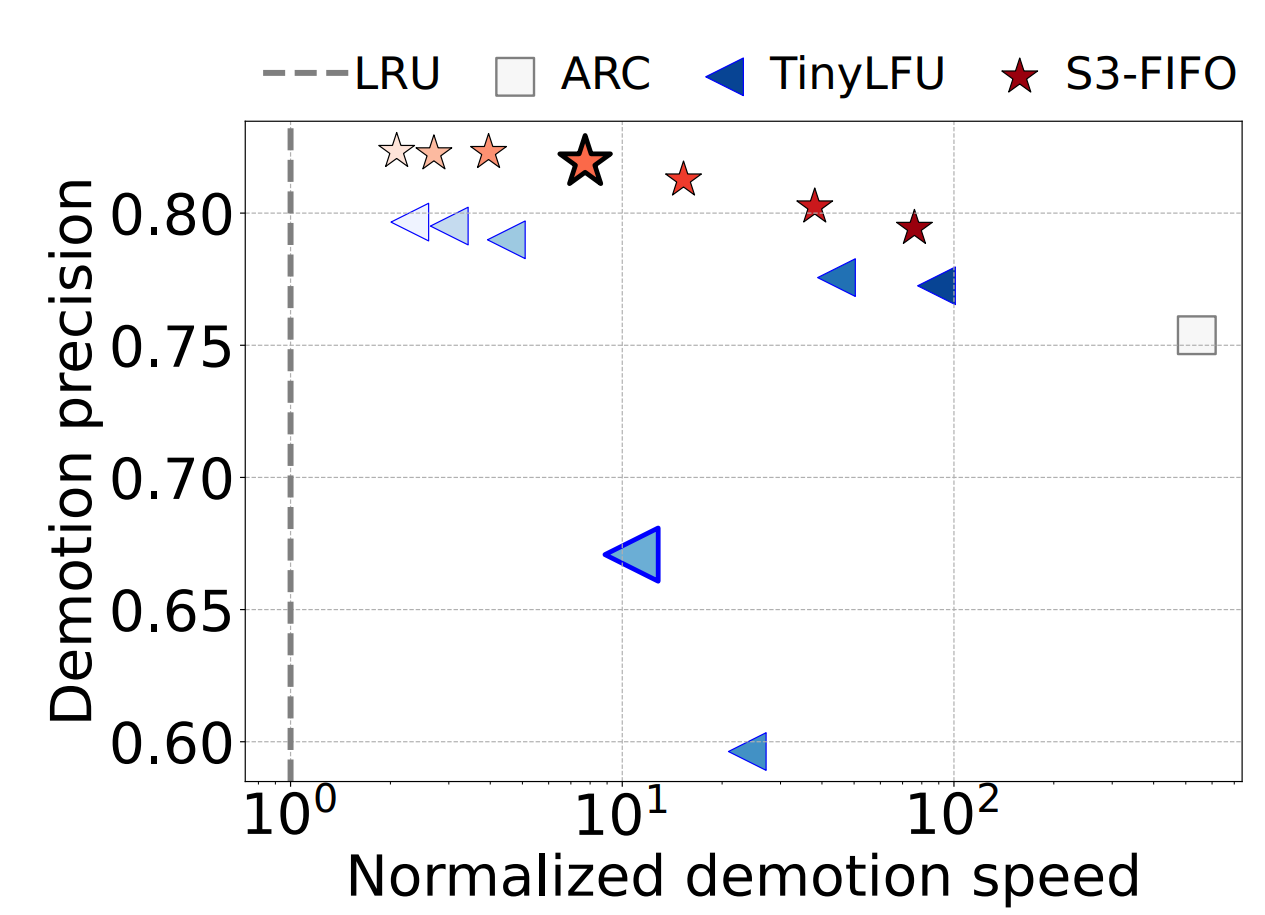
因为大部分生产数据的Cache访问序列大致符合 Zipf 分布，此图使用不同参数 Zipf分布访问序列展示了当不同Cache容量时，一次命中奇迹的比例。通过此图我们可以看出：当Cache容量占整个访问序列数据的四分之一以下时，一次命中奇迹的比例是很大的。所以我们的Cache替换算法应该需要快速把这些只访问一次的数据淘汰。



相对于之前使用模拟数据来测试一次命中奇迹，此图使用了网上公开的实际生产中的访问数据序列包括Twitter公司和MSR公司。可以看出在实际生产数据中，一次命中比例还是很高的。



此图展示了不同线程数量对不同Cache替换算法的吞吐量影响。图中比较了本报告提到的S3-FIFO算法，LRU，优化后的LRU，Segcache和优化后的TinyLFU算法，可以看出S3-FIFO算法的带宽在线程数提升时，吞吐量大幅超过除了Segcache的其他Cache替换算法。Segcache是目前性能最好的Cache替换算法，但是在使用单线程时，我们也可以看出S3-FIFO的吞吐量优于Segcache。



图中通过比较LRU，ARC，TinyLFU和S3-FIFO算法将Cache中数据降低优先级的准确度，展示了在不同的降低优先级的速度上，S3-FIFO的准确度都高于其他算法。快速降低Cache中数据的优先级不仅能提高Cache的命中率，还能设计出对闪存友好的Cache，因此S3-FIFO算法在这方面是很优秀的。

## 尝试改进

本报告在基础的S3-FIFO的基础上，用rust模拟实现了一个具有动态Small FIFO大小的S3-FIFO-Dyna，设计思路是：每次插入进行插入操作时，检查一遍是否需要调整Small FIFO的大小，如果Small Queue已满，则增大Small Queue的大小，如果较长时间没有对Small Queue进行操作，则缩小Small Queue的大小。

    fn adjust\_small\_size(&mut self) {

        if self.should\_increase\_small() {

            self.small\_size = std::cmp::min(self.small\_size + 1, self.small\_max\_size);

        } else if self.should\_decrease\_small() {

            self.small\_size = std::cmp::max(self.small\_size - 1, self.small\_min\_size);

        }

    }

    // 定义何时增加small队列大小的条件

    fn should\_increase\_small(&self) -> bool {

        self.small.len() == self.small\_size

    }

    // 定义何时减少small队列大小的条件

    fn should\_decrease\_small(&self) -> bool {

        self.small\_operated && self.small\_size > self.small\_min\_size

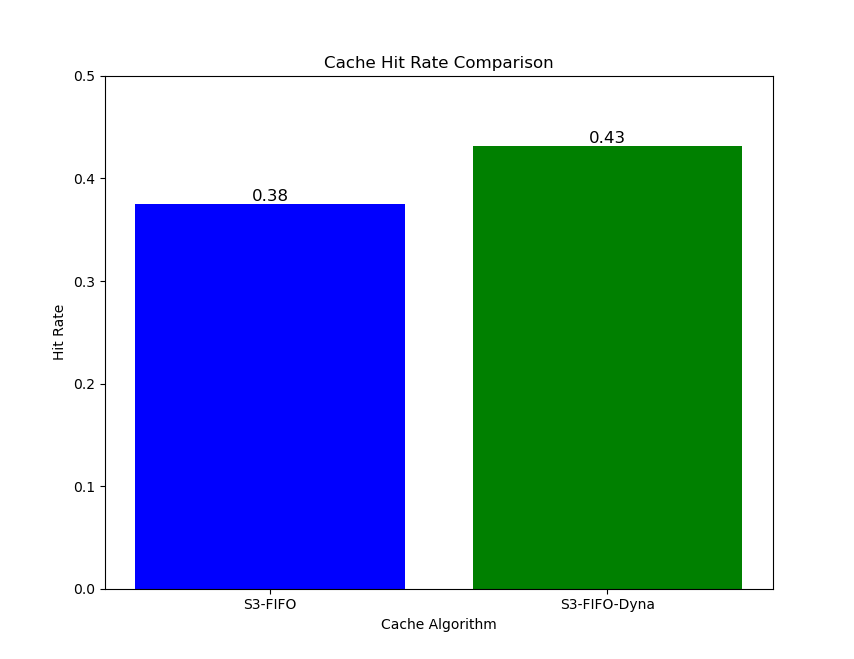
    }

对于随机的访问序列，在原有的S3-FIFO的命中率为：

1876/4995 = 0.3755755755755756

修改后的S3-FIFO-Dyna 命中率为：

2156/4995 = 0.43163163163163165



可以看出当动态调整Small Queue大小时，是有可能优化Cache的命中率的。

# 总结

本报告涉及对SOSP'23（ACM Symposium on Operating Systems Principles 2023）中提出的 "FIFO Queues are All You Need for Cache Eviction" 论文的研究与复现。这篇论文由Juncheng Yang等人撰写，介绍了一种名为S3-FIFO的新型缓存替换算法。该算法基于FIFO（先进先出）原则，具有简单、快速、可扩展和对闪存友好的特点。S3-FIFO算法通过三个静态FIFO队列——Small FIFO、Main FIFO和Ghost FIFO——的结合，优化了缓存管理，尤其是在小容量缓存环境下的表现显著。

本报告不仅复现了S3-FIFO算法，并使用CacheSim模拟器验证了其有效性和效率，还尝试对原算法进行改进。通过使用Rust语言实现S3-FIFO，并对其进行调整，报告展示了算法的可行性和高效性，同时也提出了一些潜在的改进思路。这些改进主要集中在动态调整Small FIFO队列大小上，以适应不同的使用模式和需求。

综上所述，S3-FIFO算法在缓存替换领域提供了一种新的视角和有效的解决方案，显示出了其在理论和实际应用中的广泛潜力。同时，对该算法的进一步探索和改进，展示了在现代计算环境下优化缓存管理的重要性和可能性。