|  |  |
| --- | --- |
| 学号 | 2017202110120 |

**海量存储技术课程论文**

|  |
| --- |
| **浅析“Facebook基于闪存的高级照片缓存”** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 网络空间安全 |
| 学 生 姓 名 ： | 王宇 |
| 指 导 教 师 ： | 何水兵 教授 |

摘 要

Facebook在其照片高速缓存栈广泛使用闪存设备。Facebook上一个高效的照片告诉缓存的关键设计挑战是其工作量：对高级的缓存算法来说，很多小的随机写入都是因为插入高速缓存未命中的内容或更新高速缓存命中的内容来产生的。闪存设备的FTL在这样的工作量下表现不佳，降低了吞吐量并减少了设备的使用寿命。现有的应对策略没有充分利用闪存设备上的空间，牺牲了高速缓存的容量，或者仅限于像FIFO这样的简单缓存算法一样牺牲了命中率。

Linpeng Tang等人利用新颖的受限插入优先级队列（RIPQ）框架克服了这些限制，该框架支持高缓存大小，高吞吐量和长设备使用寿命的高级缓存算法。RIPQ聚合小型随机写入，共同（协同）定位相同优先级的内容，并延迟更新内容以进一步减少设备开销。Linpeng Tang等人展示了两种高级的缓存算法，分段LRU和贪婪双尺寸频率（Segmented-LRU and Greedy-Dual-Size-Frequency），可以很容易地用RIPQ来实现。Linpeng Tang等人对Facebook的照片追踪的评估表明，这些在RIPQ上运行的算法比当前的FIFO系统提高了约20％的比例，产生了低开销，并且实现了高吞吐量。

目 录

[1 对象存储（Object-based Storage）概述 1](#_Toc500608380)

[1.1 主流网络存储架构 1](#_Toc500608381)

[1.2 对象存储架构 1](#_Toc500608382)

[1.2.1 对象 1](#_Toc500608383)

[1.2.2 对象存储设备 2](#_Toc500608384)

[1.2.3 元数据服务器 2](#_Toc500608385)

[1.2.4 对象存储系统的客户端 3](#_Toc500608386)

[1.2.5 对象存储文件系统的关键技术 3](#_Toc500608387)

[2 Facebook照片缓存 5](#_Toc500608388)

[2.1 概述 5](#_Toc500608389)

[2.2 Facebook 多级缓存架构 5](#_Toc500608390)

[2.3 Facebook 缓存的命中率提高方法 6](#_Toc500608391)

[3 闪存转换层（Flash Translation Layer） 7](#_Toc500608392)

[3.1 概述 7](#_Toc500608393)

[3.2 FTL内部mapping的机制 7](#_Toc500608394)

[4 RIPQ: Advanced Photo Caching on Flash for Facebook 10](#_Toc500608395)

[4.1 Introduction 10](#_Toc500608396)

[4.2 Background & Motivation 11](#_Toc500608397)

[4.3 Flash Performance Study 12](#_Toc500608398)

[4.3.1 Random Write Experiments 13](#_Toc500608399)

[4.3.2 Sequential Write Experiment 14](#_Toc500608400)

[4.4 RIPQ 14](#_Toc500608401)

[4.4.1 Priority Queue Abstraction 14](#_Toc500608402)

[4.4.2 Overall Design 16](#_Toc500608403)

[4.4.3 Implementing Caching Algorithms 18](#_Toc500608404)

[4.4.4 Implementation of Basic Operations 19](#_Toc500608405)

[5 总结 23](#_Toc500608406)

[参考文献 25](#_Toc500608407)

# 对象存储（Object-based Storage）概述

## 主流网络存储架构

存储局域网(SAN)和网络附加存储(NAS)是目前两种主流网络存储架构，而对象存储（Object-based Storage）是一种新的网络存储架构，基于对象存储技术的设备就是对象存储设备（Object-based Storage Device）简称OSD。1999年成立的全球网络存储工业协会(SNIA)的对象存储设备(Object Storage Device)工作组发布了ANSI的X3T10标准。总体上来讲，对象存储（Object-Based Storage, OBS)综合了NAS和SAN的优点，同时具有SAN的高速直接访问和NAS的分布式数据共享等优势，提供了具有高性能、高可靠性、跨平台以及安全的数据共享的存储体系结构。

SAN存储架构采用SCSI 块I/O的命令集，通过在磁盘或FC（Fiber Channel）级的数据访问提供高性能的随机I/O和数据吞吐率，它具有高带宽、低延迟的优势，在高性能计算中占有一席之地，如SGI的CXFS文件系统就是基于SAN实现高性能文件存储的，但是由于SAN系统的价格较高，且可扩展性较差，已不能满足成千上万个CPU规模的系统。

NAS存储架构采用NFS或CIFS命令集访问数据，以文件为传输协议，通过TCP/IP实现网络化存储，可扩展性好、价格便宜、用户易管理，如目前在集群计算中应用较多的NFS文件系统，但由于NAS的协议开销高、带宽低、延迟大，不利于在高性能集群中应用。

## 对象存储架构

对象存储架构的核心是将数据通路（数据读或写）和控制通路（元数据）分离，并且基于对象存储设备（Object-based Storage Device，OSD）构建存储系统，每个对象存储设备具有一定的智能，能够自动管理其上的数据分布。对象存储结构由对象、对象存储设备、元数据服务器、对象存储系统的客户端四节组成。

### 对象

对象是系统中数据存储的基本单位，每个Object是数据和数据属性集的综合体，数据属性可以根据应用的需求进行设置，包括数据分布、服务质量等。在传统的存储系统中用文件或块作为基本的存储单位，块设备要记录每个存储数据块在设备上的位置。Object维护自己的属性，从而简化了存储系统的管理任务，增加了灵活性。Object的大小可以不同，可以包含整个数据结构，如文件、数据库表项等。在存储设备中，所有对象都有一个对象标识，通过对象标识OSD命令访问该对象。通常有多种类型的对象，存储设备上的根对象标识存储设备和该设备的各种属性，组对象是存储设备上共享资源管理策略的对象集合等。

### 对象存储设备

每个对象存储设备（OSD）都是一个智能设备，具有自己的存储介质、处理器、内存以及网络系统等，负责管理本地的Object，是对象存储系统的核心。OSD同块设备的不同不在于存储介质，而在于两者提供的访问接口。OSD的主要功能包括数据存储和安全访问。目前国际上通常采用刀片式结构实现对象存储设备。OSD提供三个主要功能：

（1） 数据存储。OSD管理对象数据，并将它们放置在标准的磁盘系统上，OSD不提供块接口访问方式，Client请求数据时用对象ID、偏移进行数据读写。

（2） 智能分布。OSD用其自身的CPU和内存优化数据分布，并支持数据的预取。由于OSD可以智能地支持对象的预取，从而可以优化磁盘的性能。

（3） 每个对象元数据的管理。OSD管理存储在其上对象的元数据，该元数据与传统的inode元数据相似，通常包括对象的数据块和对象的长度。而在传统的NAS系统中，这些元数据是由文件服务器维护的，对象存储架构将系统中主要的元数据管理工作由OSD来完成，降低了Client的开销。

### 元数据服务器

元数据服务器（Metadata Server，MDS）控制Client与OSD对象的交互，为客户端提供元数据，主要是文件的逻辑视图，包括文件与目录的组织关系、每个文件所对应的OSD等。主要提供以下几个功能：

（1） 对象存储访问。MDS构造、管理描述每个文件分布的视图，允许Client直接访问对象。MDS为Client提供访问该文件所含对象的能力，OSD在接收到每个请求时将先验证该能力，然后才可以访问。

（2） 文件和目录访问管理。MDS在存储系统上构建一个文件结构，包括限额控制、目录和文件的创建和删除、访问控制等。

（3） Client Cache一致性。为了提高Client性能，在对象存储系统设计时通常支持Client方的Cache。由于引入Client方的Cache，带来了Cache一致性问题，MDS支持基于Client的文件Cache，当Cache的文件发生改变时，将通知Client刷新Cache，从而防止Cache不一致引发的问题。

### 对象存储系统的客户端

为了有效支持Client支持访问OSD上的对象，需要在计算节点实现对象存储系统的Client。现有的应用对数据的访问大节都是通过POSIX文件方式进行的，对象存储系统提供给用户的也是标准的POSIX文件访问接口。接口具有和通用文件系统相同的访问方式，同时为了提高性能，也具有对数据的Cache功能和文件的条带功能。同时，文件系统必须维护不同客户端上Cache的一致性，保证文件系统的数据一致。文件系统读访问流程：

（1）客户端应用发出读请求;

（2）文件系统向元数据服务器发送请求，获取要读取的数据所在的OSD;

（3）然后直接向每个OSD发送数据读取请求；

（4） OSD得到请求以后，判断要读取的Object，并根据此Object要求的认证方式，对客户端进行认证，如果此客户端得到授权，则将Object的数据返回给客户端；

（5）文件系统收到OSD返回的数据以后，读操作完成。

### 对象存储文件系统的关键技术

传统的存储结构元数据服务器通常提供两个主要功能。

（1）为计算结点提供一个存储数据的逻辑视图（Virtual File System，VFS层），文件名列表及目录结构。

（2）组织物理存储介质的数据分布（inode层）。对象存储结构将存储数据的逻辑视图与物理视图分开，并将负载分布，避免元数据服务器引起的瓶颈（如NAS系统）。元数据的VFS节通常是元数据服务器的10％的负载，剩下的90％工作（inode节）是在存储介质块的数据物理分布上完成的。在对象存储结构，inode工作分布到每个智能化的OSD，每个OSD负责管理数据分布和检索，这样90%的元数据管理工作分布到智能的存储设备，从而提高了系统元数据管理的性能。另外，分布的元数据管理，在增加更多的OSD到系统中时，可以同时增加元数据的性能和系统存储容量。

对象存储体系结构定义了一个新的、更加智能化的磁盘接口OSD。OSD是与网络连接的设备，它自身包含存储介质，如磁盘或磁带，并具有足够的智能可以管理本地存储的数据。计算结点直接与OSD通信，访问它存储的数据，由于OSD具有智能，因此不需要文件服务器的介入。如果将文件系统的数据分布在多个OSD上，则聚合I/O速率和数据吞吐率将线性增长，对绝大多数Linux集群应用来说，持续的I/O聚合带宽和吞吐率对较多数目的计算结点是非常重要的。对象存储结构提供的性能是目前其它存储结构难以达到的，如ActiveScale对象存储文件系统的带宽可以达到10GB/s。

# Facebook照片缓存

## 概述

目前，Facebook总共大概有2500亿照片，提供缓存的机器达几千台，提供存储的机器达1万2千台左右。缓存在系统中的重要性不言而喻，访问存储在磁盘上的数据速度慢，且耗费很大的带宽。所以在设计缓存系统的时候，要注意几点：延时、命中率和带宽。

图2.1为Facebook 的照片缓存系统的整体架构图，可以看出，Facebook 使用的是多级缓存策略，每一级缓存完成的功能不一样。

图2.1 Facebook 照片缓存系统整体架构图



## Facebook 多级缓存架构

图2.2 Facebook 多级缓存架构，可以看出，缓存总共分为3级：

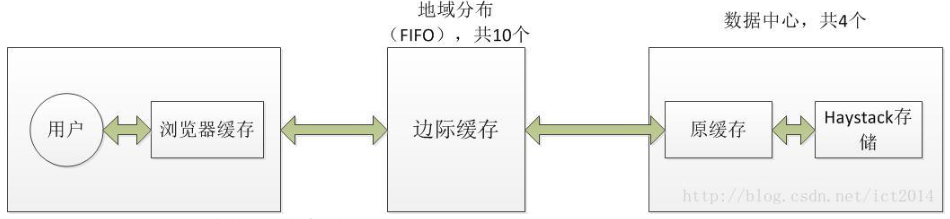
1）浏览器缓存，能够响应大约65.5%的请求。

2）边际缓存，能够响应大约19.43%的请求，边际缓存是分布在美国东中西岸10个地域，总共的缓存大小为100TB。

3）原缓存，能够响应大约4.79%的请求，分布于Facebook 在美国的4个数据中心。

上述三级缓存总共能响应大约89.72%的请求，说明Facebook 的缓存设计已经非常成功。在89.72%基础之上在进行提高已经是很具有挑战性的工作。

图2.2 Facebook 多级缓存架构



## Facebook 缓存的命中率提高方法

为了进一步提高缓存的命中率，黄琦提出了两点验证有效的解决方案：

1：S4LRU缓存算法，从图2中可以看出在10个边际缓存中采用的缓存策略是先进先出的模式（FIFO），这是一种极为简单的缓存算法。具体的实现是通过改进缓存算法，能够提高命中率。S4LRU缓存算法是LRU和LFU两种算法的结合，共分4段，属于分段缓存中的一种。

2：协同缓存，协同缓存以前一直不被认可，主要原因是由于延时、带宽等限制。由于网络日益发展，情况变化了。协同缓存可以通过“一致性哈希”等技术进行实现。

# 闪存转换层（Flash Translation Layer）

## 概述

FTL是SSD固态硬盘的软件核心技术。正因为有了FTL，NAND Flash才能被当成硬盘来使用；文件系统才可以直接把SSD当成普通块设备来使用。由于FTL是SSD设计厂商最为重要的核心技术，因此，没有一家厂商愿意透露这方面的技术信息，并且也一直没有业内的技术规范、标准存在。

FTL的重要程度在于决定了一个SSD的使用寿命、性能和可靠性。一旦FTL出现问题，那么就会导致数据读写发生错误，更为严重的是SSD盘无法被访问。优秀的FTL不仅能够提升Flash存储的使用寿命，而且还可以最优化读写性能。因此，在Flash固态存储中，FTL是一个最为重要的管理NAND Flash的软件层。

在学术界，这十年来有很多文章在讨论如果实现一个高效的FTL，例如，在有限硬件资源的环境下如何实现mapping？如何实现buffer的管理？如何实现高效的Garbage Collection？如何实现磨损均衡Wear-leveling？如何实现NAND Flash芯片之间的数据冗余（RAID on Chip）？很多算法的提出非常具有建设性和现实意义，对工业界具有很好的指导价值。

## FTL内部mapping的机制

基于NAND Flash研制存储设备是有很多挑战的，最大的问题在于NAND Flash不能像内存那样随意的写入。NAND Flash在Page页写入之前必须要将Page页所在的Block块擦除。如果研制SSD的时候严格按照这个准则，那么设计开发出来的SSD是不能用的。其一，按照这种方式进行写操作，写入的性能将会很差，其bottleneck限制在块擦除上（块擦除时间在ms级）；其二，不断的对同一Block块进行擦除操作，那么该块将会在短时间内磨损写坏，并且极易导致存储在该块上的数据丢失。因此，在设计SSD时最主要的任务就是解决NAND Flash的这种“写时擦除”问题。

技术是相通的，在90年代提出的Log-structured File System（LFS）思想和NAND Flash简直是天生一对。当初设计Log-Structured File System最主要的想法是利用机械磁盘出色的顺序写性能，避免糟糕的随机写问题。Log-structured File System在机械硬盘的时代有一定的应用局限性，问题在于采用log数据布局的方式之后，读性能变得很差。因此，只有在大块数据读写（对象存储）的环境下，log-structured File System才变得合情合理。在NAND Flash介质上，不存在机械硬盘随机读写的问题，因此，log-structured的数据布局方式不会引入任何性能问题，反而能够解决NAND Flash的“写时擦除”问题。

采用Log-structured的方式之后，NAND Flash可以采用out-of-place的数据更新方式。所有的数据更新都不会写入原来的page页，而是重映射写入一个新的Page页。在这个思路的引导，很显然所有NAND Flash的存储资源可以按照物理Page页的方式管理起来，而用户可见的空间则是一个连续逻辑Page页连接起来的地址空间。FTL的一个关键任务就是建立逻辑Page和物理Page之间的映射关系，并且在数据写入时重新分配物理Page页。在这种机制的支撑下，SSD的写性能可以大为提高，写延迟可以控制在200us的级别。

引入log-structured的机制之后，逻辑page和物理page之间存在映射关系，FTL负责物理page页的分配。考虑到每个Block块都是有擦除寿命的，因此，如果想要提升SSD的整体使用寿命，那么需要将块擦除次数均衡到所有块上去。这个工作就交给了FTL中的块分配器。均衡擦除次数这个工作其实是挺麻烦的事情，其最大的挑战在于记录每个块的擦写次数，并且这些信息需要持久化存储。

众所周知，log-structured数据布局方式最大的问题在于垃圾回收（garbage collection），由于page页从来不会被in-place-update，因此，当一个page被重映射之后，老的page页就会变成无效，等待Garbage Collection回收该页。在NAND Flash中，GC最大的挑战在于以块为单元进行擦除，而不是Page页。换句话说，GC需要将一个Block块中的所有Page页同时回收，这个限制导致GC在回收一个Block的时候会进行数据迁移操作。过多的数据迁移操作会影响SSD的使用寿命，并且会影响到整体的读写性能。因此，优化Garbage Collection成了FTL最头疼的一个问题。最容易想到的一个优化方法是将冷热数据分开存储到不同的Block块中，这样在数据回收的时候，可以尽最大可能减少有效数据的迁移。

Log-structured File System为NAND Flash的FTL设计提供了一个非常好的思路。但是，要想在SSD这样一个硬件资源非常有限的平台上实现FTL的所有功能还是很有挑战的。举个例子，log-structured的方式是需要进行Page页映射的，映射操作需要建立映射表。如果内存太小，那么对映射表的大小就提出了需求。假设一个SSD具有1TB的容量，那么如果采用4KB Page映射的方式，每个page映射需要4字节描述，那么至少需要1GB的内存容量来存放映射表。在嵌入式系统中，1GB的内存容量是庞大的。因此，为了避免过多的占用内存容量，拍脑袋可以想到Block映射的方式。假设一个Block容纳128个Page页，那么一下子可以将映射表容量缩减到原来的1/128。但是这种Block映射的方式效率实在太低，会导致大量的数据迁移，从而缩短了SSD的使用寿命。所以，为了减小映射表的容量，一个比较可行的方式是采用Hybrid映射方式。

Hybrid-level mapping的思想是将映射操作分成两级。第一级是data-log，所有数据首先写入log，当log写满之后，再将log中的数据合并至data-block；第二级是data-block，用来存放从log中合并过来的数据。对于data-log，由于数量有限，因此可以采用page-level mapping的方式；对于data-block，由于存储容量比较大，因此，可以采用block-level mapping的方式。Hybrid-level mapping可以很好的平衡内存使用和mapping效率之间关系，因此，学术界也对此提出了很多优化的方法。

其中，一个比较有意思的方法是locality aware的hybrid-level mapping思想，其原理如上图所示。从结构上讲，其大致可以分成传统Hybrid-level mapping的双层结构。写入的数据首先进入data-log。不同的地方是，data-log被分成了random-log-buffer和sequential-log-buffer。写入的数据根据locality-detector被分流至random-log-buffer或者sequential-log-buffer。其中random-log-buffer采用page页映射的方式，sequential-log-buffer直接采用block映射的方法，这样可以进一步降低内存使用量。当log-buffer中的数据满了之后，需要合并到data-block。合并的方法和传统的相同，被分成switch、partial-merge和full-merge。

FTL是NAND Flash存储的底层核心技术之一，由于NAND Flash本身存在很多问题，导致FTL的设计、实现都会存在很多的挑战。

# RIPQ: Advanced Photo Caching on Flash for Facebook

## Introduction

Facebook拥有深度分布式的照片缓存堆栈，可减少照片传输延迟和后端负载。这个堆栈使用闪存在容量上比DRAM更具优势，并且在I/O性能上比磁盘更高。

最近的一项研究[20]显示：使用一些更高级的缓存算法，如the Segmented-LRU family of algorithms，可以显著提高Facebook的照片缓存命中率。然而，在闪存上这些算法单纯的执行表现不佳。如实现了70%左右的命中率的Quadruple-Segmented-LRU,在插入高速缓存未命中的内容（约30%未命中）和更新高速缓存命中的内容（约70%命中）时产生了大量的小的随机写。这样一个随机写入繁重的工作量可能导致在现代NAND闪存设备中的闪存转换层（FTL）频繁的垃圾回收，尤其当写入的容量小的时候会导致写入放大率高、吞吐量下降以及设备使用寿命缩短[36]。

现有的解决这个问题的方法往往为FTL（over-provisioning）保留了设备空间的重要节，从而减少了垃圾收集的频率。但是，over-provisioning会降低可用缓存容量。因此，Facebook以前只使用FIFO缓存策略，以牺牲算法的优势来最大限度地提高缓存容量并避免小的随机写入。

Linpeng Tang等人的目标是设计一个闪存缓存，支持高命中率的高级缓存算法，使用大节闪存的缓存容量，并且不会导致小的随机写入。为此，Linpeng Tang等人设计并实现了新颖的受限插入优先级队列（RIPQ）框架，该框架有效地近似了闪存上的优先级队列。RIPQ为程序员提供了优先级队列的接口，Linpeng Tang等人的经验和先前的工作表明，这是实现高级缓存算法的一个方便的抽象[10,45]。

RIPQ的关键挑战和新颖之处在于如何将（确切的）优先级队列的更新转换成接近闪存的工作负载。RIPQ聚集了内存中的小型随机写入，并且只通过有限数量的闪存插入点来对齐大写入，以防止FTL垃圾收集和过多的内存缓冲。缓存中具有相似优先级的对象位于这些插入点之间。这在很大程度上保留了RIPQ之上高级缓存算法的稳定性。只有在即将被赶出（evict）的情况下，RIPQ才会延迟更新优先级的内容，从而进一步降低开销，同时不会损害保密性。结果，RIPQ以高保真度逼近优先级队列抽象，并且只在低写入放大的情况下执行合并的大对齐写入。

Linpeng Tang等人还介绍了单插入优先级队列（SIPQ）框架，该框架近似于具有单个插入点的优先级队列。SIPQ专为内存受限的环境而设计，可以使用像LRU这样的简单算法，但不适合支持更高级的算法。

RIPQ和SIPQ具有超越Facebook的照片缓存的适用性。他们应该使用高级的缓存算法进行静态内容缓存 - 即只读缓存 - 一般来说，如Netflix的基于闪存的视频缓存[38]。

Linpeng Tang等人通过实施两种高级缓存算法Segmented-LRU[26]（SLRU）和Greedy-Dual-Size-Frequency[12]（GDSF）来评估RIPQ和SIPQ，并测试他们在两层Facebook 照片缓存堆栈：与后端存储位于同一地点的Origin缓存，遍布全球的Edge缓存直接为用户提供照片。Linpeng Tang等人的评估显示，这两种算法都可以使用RIPQ和SIPQ实现更高的命中率。例如，使用RIPQ的GDSF算法将Origin缓存中的命中率提高了17-18％，导致后端I/O负载下降23-28％。

本篇论文的主要贡献包括：

闪速性能研究，确定最大吞吐量随机写入的最小尺寸的显着增加，并激励RIPQ的设计。

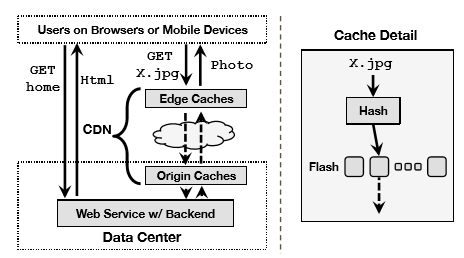
RIPQ的设计和实施是Linpeng Tang等人的主要贡献。 RIPQ是一个框架，用于实现高空间利用率，高吞吐量和长设备寿命的高级缓存算法。

Facebook上的照片追踪评估表明，RIPQ（和SIPU上的LRU）上的高级缓存算法可以实现高保真，高吞吐量和低设备开销。

## Background & Motivation

如图3.1所示，Facebook的照片服务堆栈包括两个缓存层：Edge缓存层和Origin缓存。在每个缓存站点，单个照片对象根据其URI被散列到不同的缓存机器。然后，每个缓存机器作为其对象子集的独立缓存。

图3.1 Facebook photo-serving stack



Edge缓存层包括许多独立的高速缓存，遍布全球的互联网点（POP）。Edge缓存层的主要目标除了减少用户的延迟外，还要减少发送到Facebook数据中心的流量，因此评估其有效性的指标是按字节顺序排列的命中率。Origin缓存是分布在Facebook数据中心的单个缓存，位于Edge缓存后面。它的主要目标是减少对Facebook基于磁盘的存储后端的请求，所以其有效性度量是面向对象的命中率。面对大量对象的高请求率，Edge和Origin高速缓存都配备了闪存驱动器。

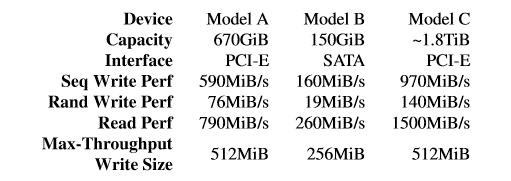
这项工作的动机是发现SLRU是一种高级的缓存算法，可以将Facebook堆栈中的按字节顺序和按对象顺序的命中率提高多达14％[20]。然而，有两个因素混淆了高级缓存算法在闪存上的朴素的实现。一方面，不同缓存位置的工作负载的最佳算法是不同的。例如，由于黄等人[20]。Linpeng Tang等人发现GDSF通过支持较小的对象（参见6.2节），在Origin高速缓存中实现了比SLRU更高的对象命中率，但是SLRU在Edge缓存中依然实现了最高的逐字节命中率。因此，许多高速缓存算法的统一框架可以大大减少工程设计工作量，并加快部署新的高速缓存策略。其次，基于闪存的硬件具有独特的性能特点，往往需要软件定制。特别是，高级的缓存算法的一个朴素的实现可能会产生大量的随机写入，通过插入内容或更新内容。

## Flash Performance Study

本节介绍了一个激励Linpeng Tang等人设计的现代闪存设备的研究。该研究侧重于强调设备上的FTL的写入工作负载，因为写入吞吐量是阻止Facebook部署高级缓存算法的瓶颈。即使对于只读缓存，写入也是工作负载的重要组成节，因为错过的内容是通过写入来插入的。在Facebook上，即使利用高级的缓存算法，最大命中率也达到了70％，至少有30％的访问是写入。

以前的研究[17,36]已经表明，小的随机写对闪存有害。特别是，Min等人[36]表明，在高空间利用率（即90％）下，随机写入大小必须大于16 MB或32 MB才能达到2012年三个代表性SSD上的峰值吞吐量，容量范围介于32 GB和64 GB之间。为了更好地了解目前的闪存设备，Linpeng Tang等人研究了三张闪存卡的性能特点，其规格和主要指标列于表4.1。所有这三款设备都是来自主要厂商的最新型号，A和C目前部署在Facebook照片缓存中。

表4.1 Flash performance summary



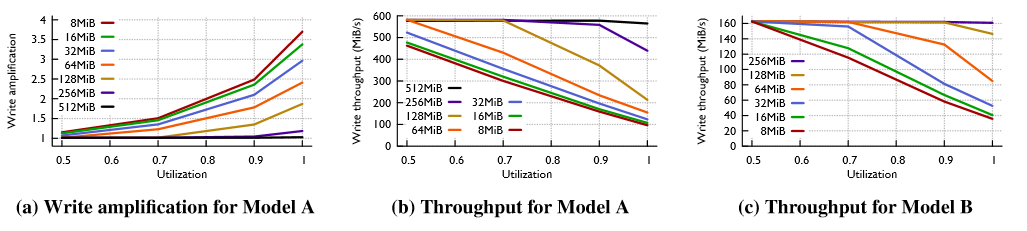
### Random Write Experiments

本小节介绍的实验探讨了随机写入性能的写入大小和设备过度配置之间的权衡空间。在这些实验中，Linpeng Tang等人使用不同的大小对设备进行分区，然后在不同的空间利用率下对齐随机写入。Linpeng Tang等人使用闪存驱动器作为原始块设备来避免文件系统开销。在每次运行之前，Linpeng Tang等人使用blkdiscard清除现有的数据，然后反复挑选一个随机对齐的位置来执行写入/覆盖。在报告最终的稳定吞吐量之前，Linpeng Tang等人向设备写入4倍的总容量数据。在每个实验中，初始吞吐量总是很高，但是当设备变满时，垃圾收集器开始工作，导致FTL写入放大和吞吐量急剧下降。

在垃圾回收期间，FTL经常向主机写入更多的数据到物理设备，而这两个写入大小之间的字节比是FTL写入放大[19]。图4.2a和图4.2b显示了在A型闪存驱动上进行的随机写入实验的FTL写放大率和设备吞吐量。这些图表明，随着写入变小或空间利用率增加，写入吞吐量急剧下降，FTL写入放大率增加。例如，在90％设备利用率下的8个MiB随机写入只能达到160MiB/s，比最大590MiB/s降低约3.7倍。Linpeng Tang等人还尝试了混合读写工作负载，并保持相同的性能趋势。特别是，50％的读取和50％的写入工作量，在90％利用率下的8个MiB随机写入导致吞吐量减少约2.3倍。高FTL写入放大也降低了设备的使用寿命，随着大容量闪存卡的擦除周期的不断减少，小随机写入的影响随着时间的推移变得更糟[5,39]。

在图4.2c中示出了在闪存驱动B上类似的吞吐量结果。但是，由于设备上缺乏监视物理写入的工具，FTL写放大功能不可用。Linpeng Tang等人的闪存驱动模型C（由于空间限制细节）的实验同样与模型A和B的结果一致。由于在写入量较小的情况下利用率较高，吞吐量较低，总共花费了超过1000个设备小时来生成图2中的数据点。

图4.2 Random write experiment on Model A and Model B



虽然Linpeng Tang等人的发现与前面的研究一致[36]，但Linpeng Tang等人惊奇地发现，在90％的设备利用率下，达到峰值随机写入吞吐量的最小写入容量已经达到256 MiB到512 MiB。由于现代闪存硬件由许多并行NAND闪存芯片组成[3]，并且所有并行芯片上的累计擦除块大小可累加达数百兆字节，所以这种大的写入大小是必要的。与供应商工程师的沟通证实了这个假设。这个约束告知了RIPQ的设计，它只发出大对齐的写入，以实现低写放大和高吞吐量。

### Sequential Write Experiment

实现持续高闪存写入吞吐量的常用方法是发出顺序写入。FTL可以将顺序写入有效地聚合到并行擦除块[30]，删除和覆盖所有并行块可以一起擦除而不写回任何有效的数据。因此，FTL写放大率可能很低，甚至完全可以避免。为了证实这一点，Linpeng Tang等人还对同样的三个闪存设备进行了顺序写入实验。Linpeng Tang等人观察到表4.1中报告的所有写入大小在128KiB以上的持续高性能。这个结果启发了SIPQ的设计，它只发出顺序写入。

## RIPQ

本节介绍RIPQ框架的设计和实现。Linpeng Tang等人展示了它如何接近闪存设备上的优先级队列抽象，展示其实现细节，然后证明它有效地支持高级缓存算法。

### Priority Queue Abstraction

Linpeng Tang等人的经验和以前的研究[10,45]表明，优先级队列是一个普遍的抽象，自然支持各种高级的缓存策略。RIPQ通过维护其内部近似优先级队列中的内容来提供抽象，并允许通过三个基元进行缓存操作：

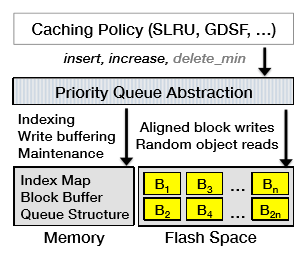
• **insert(x, p)**: insert a new object x with priority value p.

• **increase(x, p)**: increase the priority value of x to p.

• **delete-min()**: delete the object with the lowest priority.

对象的优先级值表示它对缓存算法的效用。在命中时，调用增加来调整访问对象的优先级。顾名思义，RIPQ限制优先级调整只能增加。这个约束简化了RIPQ的设计，并且仍然允许实现几乎所有的缓存算法。在未命中时，插入被调用来添加访问的对象。当通过插入触发高速缓存驱逐时，隐式地调用Delete-min来移除具有最小优先级值的对象。图4.3显示了使用优先级队列抽象实现的缓存解决方案的体系结构，其中RIPQ的组件以灰色突出显示。这些组件对于避免小型随机写入工作负载至关重要，这可以通过优先队列的单纯实现来生成。

图4.3 Advanced caching policies with RIPQ



**Absolute/Relative Priority Queue** 使用RIPQ的高速缓存设计人员可以根据访问时间、访问频率、大小以及许多其他因素（取决于高速缓存策略）来指定其内容的优先级。尽管传统的优先级队列通常使用保持固定的绝对优先级值，但RIPQ在不同的相对优先级值接口上运行。在相对优先级队列中，对象的优先级是[0,1]范围内的一个数字，表示该对象相对于其余队列的位置。例如，如果对象i的相对优先级为0.2，那么队列中的20％的对象的优先级值低于i，并且它们的位置更靠近尾部。

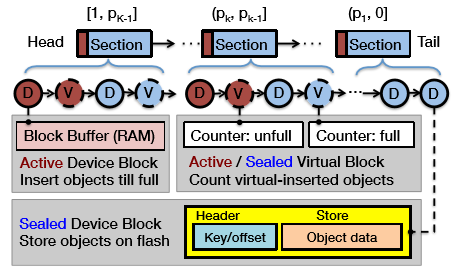
调用增加时，对象的相对优先级被明确地改变。对象的相对优先级也随着其他对象被插入队列头部而被隐含地减少。例如，如果一个对象j被插入的优先级为0.3，那么所有优先级≤0.3的对象将被推向尾部，并且它们的优先级值会被隐式地降低。

许多算法，包括SLRU系列，都可以通过相对优先级队列接口轻松实现。包括GDSF系列在内的其他算法，则需要绝对的优先级接口。为了支持这些算法，RIPQ将绝对优先级转换为相对优先级。

### Overall Design

RIPQ是一个框架，可以将优先级队列操作转换为具有大写操作的闪存友好型工作负载。图4.4给出了图4.3中突出显示的RIPQ组件的详细说明，不包括索引图。

图4.4 Overall structure of RIPQ



**Index Map** 索引映射是内存中的哈希表，它将所有对象的键与其元数据相关联，包括其在RAM中的位置或闪存，大小和块ID。

在Linpeng Tang等人的系统中，每个条目都是大约20个字节，RIPQ增加了2个字节来存储对象的虚拟块ID。考虑到闪存卡的容量和平均对象的大小，一个缓存机器中有大约5000万个对象，总计约为1GiB。

**Queue Structure** RIPQ的主要队列结构由K节组成，而每节又是由块组成的。节将插入点定义到队列中，块是写入闪存的数据单元。相对优先权值范围被分割成对应于区段[K，K，K]的区间：[1，pK-1]，...，（pk，pk-1]，...，（p1,0）。当一个对象以优先级p被插入到队列中时，它被放置在其范围包含p的区段的头部。例如，在一个段落对应于[1,0.7]，（0.7,0.3]和（0.1,0]的队列中，优先级值为0.5的对象将被插入到第二节的头部。与相关优先级队列类似，当一个对象被插入N个对象的队列中时，优先级为q的同一个或下一个区段中的任何对象被从优先级q隐式降级到qN/(N+1)。隐式降级捕捉许多缓存算法（包括SLRU和GDSF）的动态变化：当新对象插入到队列中时，旧对象的优先级逐渐降低，并在优先级达到0时从缓存中逐出。

RIPQ近似于优先级队列抽象，因为它的设计限制了数据可以插入的地方。插入点数K表示插入精度和内存消耗之间在RIPQ中的关键设计折衷。每个节具有O（1/K）的大小，因此更大的K值导致较小的节，进而有了较高的插入精度。但是，由于每个活动块都在RAM中进行缓冲，直到其满溢到闪存，因此RIPQ的内存消耗与K成正比。Linpeng Tang等人的实验显示K = 8确保RIPQ达到类似于精确算法的命中率，并且Linpeng Tang等人在Linpeng Tang等人的实验中使用这个值。256MiB设备模块可以转换成2GiB的适中内存。

**Device and Virtual Blocks** 如图4.4所示，每个节包括一个活动设备块，一个活动虚拟块和一个密封设备/虚拟块的有序列表。一个活动的设备块接受新对象的插入并将它们缓存在内存中，即块缓冲器。当这个活动设备块充满时密封，流向闪存，并过渡进一个密封的设备块。为了避免重复数据，RIPQ在优先级增加时延迟更新对象的位置，并使用虚拟块跟踪对象将被移动的位置。每个节头部的活动虚拟块接受具有更高优先级的虚拟更新对象。当一个节的活动设备块被密封时，RIPQ也会将活动虚拟块转换为密封的虚拟块。虚拟更新是内存中的唯一操作，它在索引映射中设置对象的虚拟块ID，增加目标虚拟块的大小计数器，并减少对象原始块的大小计数器。

与密封装置块相关联的所有对象都存储在闪存的连续空间中。在每个块中，一个标题将在标题后面的数据中记录所有的对象键（关键字）和它们的偏移量。如前所述，更新的对象在索引映射中标有其目标虚拟块ID。在逐出密封的设备块时，块头被检查以确定块中的所有对象。在索引图中查找对象以查看它们的虚拟块ID是否被设置，即，在插入之后它们的优先级被增加。如果是这样的话，那么RIPQ将这些对象重新插入到它们的虚拟块所代表的优先级中。对象移动到活动的设备块中，并删除其对应的虚拟对象。由于更新后的对象将不会被写入，直到旧对象即将被驱逐，RIPQ最多可以保留每个对象的一个副本，并避免重复。另外，延迟更新还允许RIPQ在插入和重新插入之间将所有优先级更新合并到一个对象中。

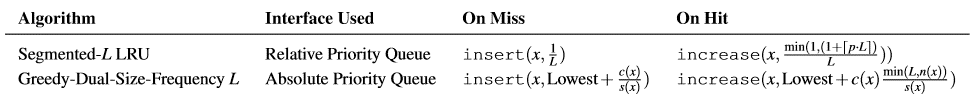
设备块在RAM（活动）或闪存（密封）的大型连续空间占用较大的缓冲区。相比之下，虚拟块只驻留在内存中，而且非常小。每个虚拟块仅包括元数据，例如其唯一ID，对象初始的计数以及这些对象的总字节大小。

Naive Design 闪存的优先级队列的一个朴素的设计是将对象的位置固定在闪存上直到它被驱逐。这种设计避免了任何写入优先级更新的闪存，但不会将对象的位置与其优先级对齐。因此，被驱逐对象的闪存空间是不连续的，FTL必须通过复制这些分散对象才能重新使用空间，从而产生显着的FTL写入放大效果。RIPQ通过将具有相似优先级的对象分组到大块中，并在块级别上执行写入和逐出，以及使用延迟更新来避免写入更新，从而避免了此问题。

### Implementing Caching Algorithms

为了证明RIPQ的灵活性，Linpeng Tang等人实现了两个先进缓存算法系列用于评估：Segmented-LRU[26]和Greedy-Dual-Size Frequency[12]，这两个产品都会为Facebook照片工作量带来重大的缓存性能提升。表4.2列出了实施情况的总结。

表4.2 SLRU and GDSF with the priority queue interface provided by RIPQ



**Segmented-LRU** Segmented-L LRU (S-L-LRU) 维护同等大小的L个LRU缓存。在未命中时，一个对象被插入到第L个（最后一个）LRU缓存的头部。在命中时，一个对象被提升到前一个LRU缓存的头部，即，如果它在子缓存l中，它将被提升到最大第（l-1,1）LRU缓存的头部。从第l个缓存中逐出的对象将移到第（l+1）个缓存的头部，从最后一个缓存中逐出的对象将从整个缓存中逐出。该算法被证明能够为Facebook Edge和Origin缓存提供显着的缓存命中率改进。

使用相对优先级队列接口直接实现这一系列缓存算法。在未命中时，该对象以优先级值1/L插入，等于第L个缓存的头部。在命中时，基于访问对象的现有优先级p，RIPQ将它从第⌈(1-p)\*L⌉缓存提升到前一个缓存的头部，使用新的更高优先级min（1,(1+⌈p\*L⌉)/L）。通过相对优先级队列抽象，当另一个对象被插入/更新为更高优先级时，对象的优先级自动降低。当一个对象被插入到第l个LRU缓存的头部时，第1至第L缓存中的所有对象被降级，而这些缓存尾部的将被降级到下一个较低优先级缓存，或者如果它们在最后一个L缓存中则被逐出 - SLRU的动态性被相对优先级队列接口正确捕获。

**Greedy-Dual-Size-Frequency** Greedy-Dual-Size算法提供了一个原则性的方法，通过利用较小的对象来折衷增加的逐个命中率和逐个字节的命中率。对于Origin缓存来说，它比SLRU（第2节）实现了更高的面向对象的命中率，并且因为Origin缓存的主要目的是为了保护后端存储免受过多IO请求的支持，所以该用例更受欢迎。Greedy-Dual-Size Frequency（GDSF）通过考虑频率来改进GDS（通用数据库系统）。在GDSF中，Linpeng Tang等人将对象x的优先级更新为Lowest + c(x)\*n/s(x)，因为它被插入到缓存中，其第n次访问的优先级就是Lowest + c(x)\*n/s(x)，其中c(x)是x上缺失的程序员定义的惩罚，Lowest是当前优先级队列的最低优先级值，s(x)是对象的大小。Linpeng Tang等人使用GDSF的变体，将对象频率的最大值限制为L。L与SLRU中的段数相似。它可以防止经常访问的对象的优先级值被放大，并更好地适应动态工作负载。因此，Linpeng Tang等人的GDSF算法变体的更新规则是p(x)←Lowest+c(x)\*min(L,n)/s(x) 。因为Linpeng Tang等人正在最大化面向对象的命中率，Linpeng Tang等人为所有对象设置c(x)= 1。GDSF使用绝对优先级队列接口。

**Limitations** RIPQ还支持许多其他高级缓存算法，如LFU，LRFU[28]，LRU-k[40]，LIRS[24]，SIZE[1]，但有一些显着的例外是不能用单个RIPQ实现的，例如，MQ[48]和ARC[34]。这些算法涉及多个队列，因此不能用一个RIPQ实现。扩展Linpeng Tang等人的设计以支持在同一硬件上共存多个RIPQ，这是Linpeng Tang等人未来的方向之一。更新界面的限制更大，只允许增加优先级值。降低访问对象优先级的算法，如MRU[13]，不能用RIPQ来实现。MRU被设计用来处理大型数据集的扫描，不适用于Linpeng Tang等人的用例。

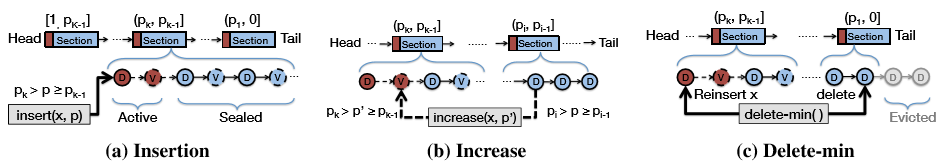
RIPQ不支持删除/覆盖操作，因为这些操作对于照片等静态内容不需要。但是，它们对于一个通用的读写缓存是必要的，增加对它们的支持也是Linpeng Tang等人未来的方向之一。

### Implementation of Basic Operations

RIPQ使用上述数据结构实现了三个常规优先级队列的操作。

**Insert(x,p)** RIPQ将对象插入到包含p的k节的活动（active）设备块中，即pk> p≥pk-1。写入将被缓冲，直到该活动块被封闭。图4.5a显示了一个插入。

图4.5 Insertion, and increase, and delete-min operations in RIPQ



**Increase(x,p)** RIPQ避免移动已经驻留在队列中的设备块中的对象x。相反，RIPQ实际上将x插入到包含p的k节的活动虚拟块中，即pk>p≥pk-1，并从逻辑上将其从当前位置移除。因为Linpeng Tang等人记住了索引散列表中的对象条目中的虚拟块ID，所以通过设置/重置对象条目的虚拟块ID并相应地更新块和段的大小计数器来简单地实现这些步骤。在此操作过程中，不会执行闪存的读/写操作。图4.5b显示了一个更新。

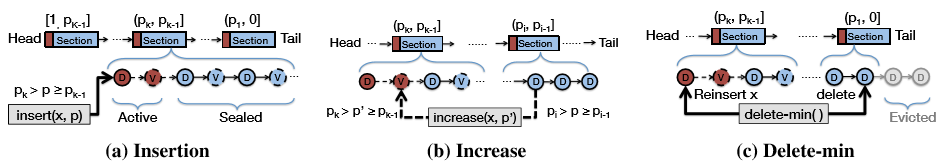
**Delete-min()**  Linpeng Tang等人维持了一些保留的闪存块，用于在设备块密封时流向设备块的RAM缓冲区。当保留块的数量低于此阈值时，将隐式调用Delete-min（）操作以释放闪存空间。如图4.5c所示，在操作期间，队列中最低优先级的块被从队列中逐出。但是，由于该块中的某些对象可能已更新到队列中较高的位置，因此需要重新插入它们以保持其正确的优先级。重新插入（1）从块头读出该块中的所有对象的关键字，（2）查询索引结构以找出对象x是否有虚拟位置，如果有，（3）找到该虚拟块的对应节k，并将数据复制到RAM中该节的活动设备块，（4）最后将虚拟块的实际位置设置为空白。Linpeng Tang等人称之为虚拟更新的整个过程实现。

这些重新插入有助于保持缓存算法的安全性，但会导致对闪存额外的写入。这些额外的写入会导致实现写入放大，这是主机发出的写入与插入缓存未命中所需的字节比率。通过跳过优先级小于给定阈值的虚拟对象的实体化，RIPQ可以显式交换较低的缓存算法的保真度以降低写入放大率，例如在队列的最后5％中。该阈值是逻辑占用参数θ(0<θ<1）。

**Internal operations** RIPQ必须有足够的插入点：太少导致精度低，太多会导致高内存使用率。为避免出现这些情况，RIPQ会在分区过大时分割一个分区，并在其总大小过小时合并连续分区。这与B树如何拆分/合并节点来控制节点的大小和树的深度是相似的。

参数α以合理的方式控制RIPQ的段数。α在（0,1）中，并确定节的平均大小。当RIPQ的相对大小（即基于对象计数或字节大小的比率）达到2α时，RIPQ将分割一个节。例如，如果α= 0.3，则[0.4,1.0]的一节将分别分割为[0.4,0.7）和[0.7,1.0]的两个节，如图4.6a所示。如果RIPQ的大小之和小于α，RIPQ合并两个连续的节，如图4.6b所示。这些操作确保最多有⌈2/α⌉节，并且每个节不大于2α。

图4.6 RIPQ internal operations



没有数据在分割或合并时在闪存上被移动。拆分一个节将创建一个具有写入缓冲区和新的活动虚拟块的新的活动设备块。合并两个节合并它们的两个活动设备块：一个的写入缓冲区被复制到另一个的写入缓冲区。分离经常发生，并且是如何将新的节添加到队列中，因为尾部的节中的对象被逐块驱逐。合并是非常罕见的，因为它要求两个连续节的总大小从2α（α是分割后的新节的大小）缩小到α来触发合并。优先级队列API提供的每个操作的合并复杂度仅为O（1/αM），其中M是块的数量。

**Supporting Absolute Priorities** 执行插入和更新时，LFU、SIZE和Greedy-Dual-Size等缓存算法需要使用绝对优先级值。RIPQ通过映射数据结构支持绝对优先级，将其转换为相对优先级。数据结构保持一个动态直方图，支持绝对优先级值的插入/删除，并在给定绝对优先级时返回近似分位数，用作内部相对优先级值。

直方图由一组容器组成，Linpeng Tang等人根据它们的相对大小动态合并/拆分容器，类似于Linpeng Tang等人在RIPQ中合并/拆分节的方式。Linpeng Tang等人可以使用比这个动态直方图更多的节来使用更多的容器，并且实现更高的转换准确度，例如，κ= 100个分容器，而RIPQ仅使用K = 8个分区，因为分容器只包含绝对优先级值，并且不需要像节那样需要大的专用RAM缓冲区。为了将优先级值插入直方图，可以进一步应用一致的密钥采样，以减少其内存消耗和插入/更新的复杂性。

# 总结

本报告首先分别介绍了对象存储、Facebook的照片缓存于和闪存的FTL层的相关知识，并基于这些知识分析总结了Linpeng Tang等人设计实现的Facebook基于闪存的高级照片缓存框架RIPQ。

Linpeng Tang等人设计并实现了新颖的受限插入优先级队列（RIPQ）框架，该框架有效地近似了闪存上的优先级队列。

RIPQ的关键挑战和新颖之处在于如何将（确切的）优先级队列的更新转换成接近闪存的工作负载。RIPQ聚集了内存中的小型随机写入，并且只通过有限数量的闪存插入点来对齐大写入，以防止FTL垃圾收集和过多的内存缓冲。缓存中具有相似优先级的对象位于这些插入点之间。这在很大程度上保留了RIPQ之上高级缓存算法的稳定性。只有在即将被赶出的情况下，RIPQ才会延迟更新优先级的内容，从而进一步降低开销，同时不会损害保密性。结果，RIPQ以高保真度逼近优先级队列抽象，并且只在低写入放大的情况下执行合并的大对齐写入。

Linpeng Tang等人通过实施两种高级缓存算法SLRU和GDSF来评估RIPQ，并测试其在两层Facebook 照片缓存堆栈：与后端存储位于同一地点的Origin缓存，遍布全球的Edge缓存直接为用户提供照片。Linpeng Tang等人的评估显示，这两种算法都可以使用RIPQ实现更高的命中率。例如，使用RIPQ的GDSF算法将Origin缓存中的命中率提高了17-18％，导致后端I/O负载下降23-28％。

通过对RIPQ: Advanced Photo Caching on Flash for Facebook这篇文章的学习，了解了Facebook现有的照片缓存的机制优点及不足，了解了闪存的优缺点以及闪存转换层的相关知识，并学习了Linpeng Tang等人设计的用于实现高空间利用率，高吞吐量和长设备寿命的高级缓存算法的RIPQ框架。为今后的研究学习增加了知识并开拓了眼界。

参考文献

[1] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal policies in network caches for World-Wide Web documents. In ACM SIGCOMM Computer Communication Review, 1996.

[2] A. Aghayev and P. Desnoyers. Log-structured cache: trading hit-rate for storage performance (and winning) in mobile devices. In USENIX INFLOW, 2013.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In USENIX ATC, 2008.

[4] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In USENIX ATC, 2013.

[5] D. G. Andersen and S. Swanson. Rethinking ﬂash in the data center. IEEE Micro, 2010.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A.Phanishayee, L.Tan, and V.Vasudevan. FAWN: A fast array of wimpy nodes. In ACM SOSP, 2009.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. Springer, 2002.

[8] L. Bouganim, B. r Jnsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In CIDR, 2009.

[9] C++11 Thread Support Library. http://en. cppreference.com/w/cpp/thread, 2014.

[10] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In USENIX USITS, 1997.

[11] B.Chazelle. The soft heap: an approximate priority queue with optimal error rate. Journal of the ACM, 2000.

[12] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In Springer HPCN, 2001.

[13] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. Algorithmica, 1986.

[14] Facebook Database Engineering Team. RocksDB, A persistent key-value store for fast storage environments. http://rocksdb.org, 2014.

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM, 1987.

[16] S. Ghemawat and J. Dean. LevelDB, A fast and lightweight key/value database library by Google. https://github.com/google/ leveldb, 2014.

[17] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a ﬂash translation layer employing demand-based selective caching of page-level address mappings. In ACM ASPLOS, 2009.

[18] J.E.Hopcroft. Data structure sand algorithms. AddisonWeely, 1983.

[19] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write ampliﬁcation analysis in ﬂash based solid state drives. In ACM SYSTOR, 2009.

[20] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In ACM SOSP, 2013.

[21] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving ﬂash-based disk cache with Lazy Adaptive Replacement. In IEEE MSST, 2013.

[22] Intel Thread Building Blocks. https://www. threadingbuildingblocks.org, 2014.

[23] D. Jiang, Y. Che, J. Xiong, and X. Ma. uCache: A Utility-Aware Multilevel SSD Cache Management Policy. In IEEE HPCC EUC, 2013.

[24] S. Jiang and X. Zhang. LIRS: an efﬁcient low inter-reference recency set replacement policy to improve buffer cache performance. In ACM SIGMETRICS, 2002.

[25] K. Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. Computer Networks, 2009.

[26] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. IEEE Computer, 1994.

[27] T. Kgil, D. Roberts, and T. Mudge. Improving NAND ﬂash based disk caches. In ACM/IEEE ISCA, 2008.

[28] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y.Cho, and C.S.Kim. LRFU:A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers, 2001.

[29] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND ﬂash memory-based storage systems. ACM SIGOPS Operating Systems Review, 2008.

[30] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based ﬂash translation layer using fully-associative sector translation. ACM Transactions on Embedded Computing Systems, 2007.

[31] C.Li, P.Shilane, F.Douglis, H.Shim, S.Smaldone, and G.Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. InUSENIXATC,2014.

[32] H.Lim, B.Fan, D.G.Andersen, and M.Kaminsky. SILT: A memory-efﬁcient, high-performance key value store. In ACM SOSP, 2011.

[33] S.Maffeis. Cache management algorithms for ﬂexible ﬁlesystems. ACM SIGMETRICS Performance Evaluation Review, 1993.

[34] N. Megiddo and D. S. Modha. ARC: A Self Tuning, Low Overhead Replacement Cache. In USENIX FAST, 2003.

[35] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. VCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In USENIX ATC, 2014.

[36] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I.Eom. SFS: Random write considered harmful in solid state drives. In USENIX FAST, 2012.

[37] D.Mituzas. Flash cache at Facebook: From2010to 2013 and beyond. https://www.facebook. com/notes/10151725297413920, 2014.

[38] Netﬂix. Netﬂix Open Connect. https://www. netflix.com/openconnect, 2014.

[39] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Improving performance and lifetime of the SSD RAID-based host cache through a log-structured approach. In USENIX INFLOW, 2013.

[40] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In ACM SIGMOD, 1993.

[41] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: software-deﬁned ﬂash for webscale internet storage systems. In ACM ASPLOS, 2014.

[42] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: a lightweight, consistent and durable storage cache. In ACM EuroSys, 2012.

[43] R. Stoica, M. Athanassoulis, R. Johnson, and A.Ailamaki. Evaluating and repairing write performance on ﬂash devices. In ACM DAMON, 2009.

[44] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ Princeton Technical Report. https://www.cs.princeton.edu/ research/techreps/TR-977-15, 2014.

[45] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. Computer Networks and ISDN Systems, 1997.

[46] J.Yang,N.Plasson,G.Gillis,andN.Talagala.Hec: improving endurance of high performance ﬂashbased cache devices. In ACM SYSTOR, 2013.

[47] N. Young. The k-server dual and loose competitiveness for paging. Algorithmica, 1994.

[48] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In USENIX ATC, 2001.