# 第一期热点报告-论文类

近年来，随着新型存储器件的应用及制造工艺的提升，存储设备所能提供的带宽及IOPS已经越来越高，然而传统linux IO软件栈是建立在慢速设备基础之上，由多个独立的软件层次组成，如图1所示，已经不能满足如今快速设备的需求，软件栈带来的时延开销在整个IO请求延时中所占比例已经越来越大，如图2 所示，现在亟需对传统linux IO栈中的某些层次或全栈进行优化乃至重新设计，以提升IO系统整体性能。

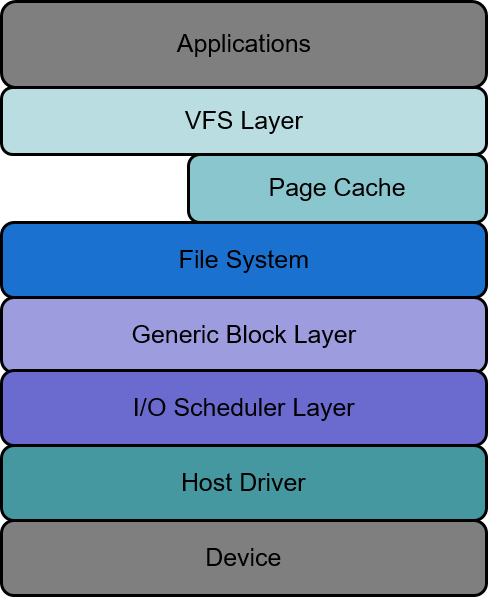


图1 linux IO栈

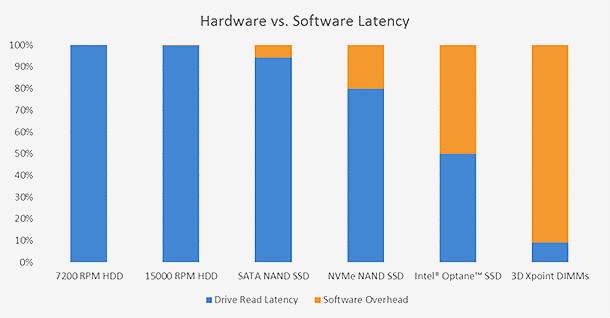


图2 软件栈延时占比

本文总结近年国际计算机相关会议论文中关于IO栈优化的若干文章，主要分为三个领域：①智能手机IO栈优化；②一般IO栈优化；③数据中心IO栈优化。每篇文章针对的问题以及优化的层次如下图2所示（有色方块代表相应文章优化的层次），

余下篇幅具体介绍每篇文章针对的问题及提出的优化方案。

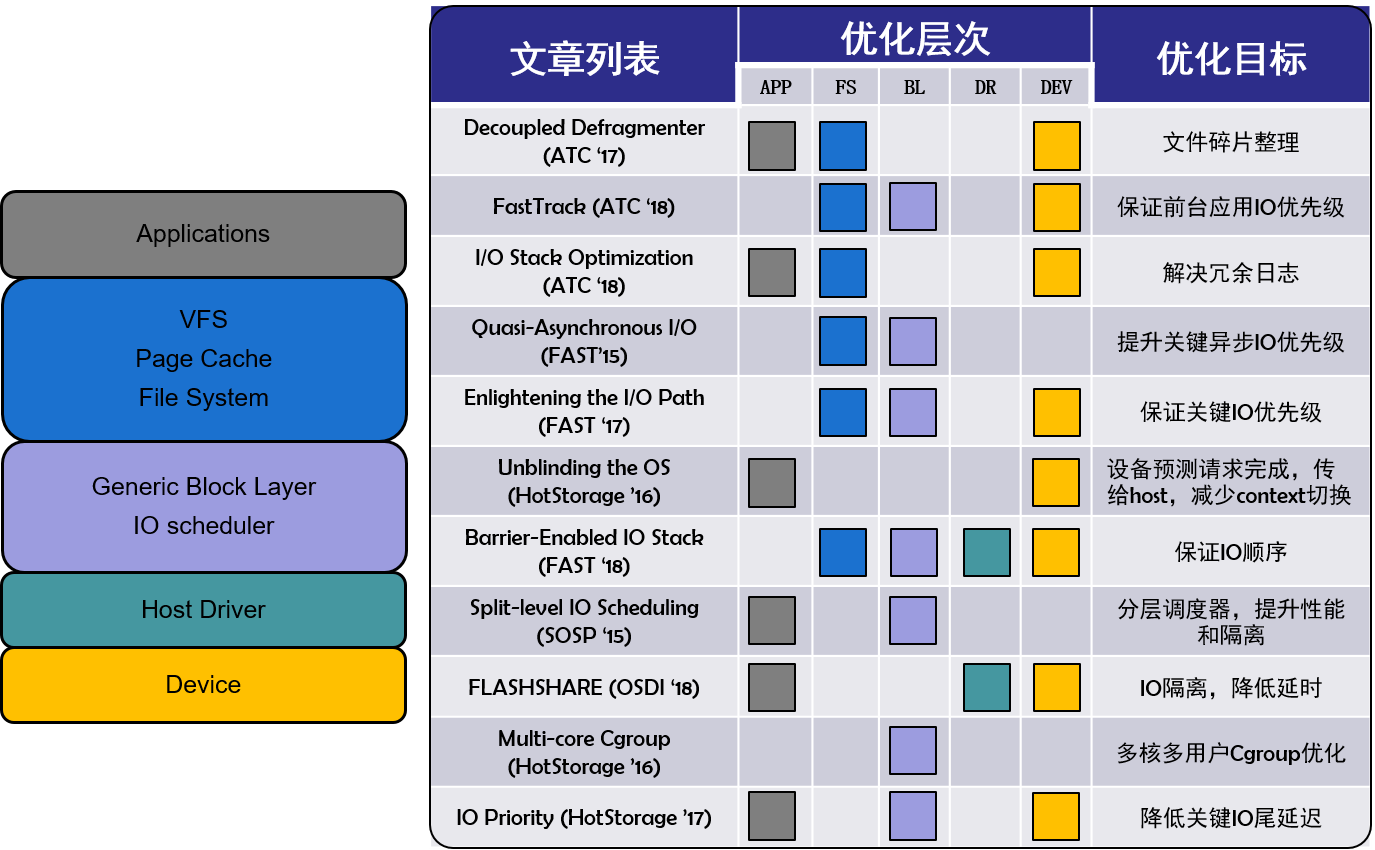


图2 IO栈优化层次

# 智能手机IO栈优化

## Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter

### 出处与作者

出处：USENIX ATC ’17

作者：**Sangwook Shane Hahn**, Seoul National University;

**Sungjin Lee**, Daegu Gyeongbuk Institute of Science and Technology;

**Cheng Ji**, City University of Hong Kong;

**Li-Pin Chang**, National Chiao-Tung University;

**Inhyuk Yee**, Seoul National University;

**Liang Shi**, Chongqing University;

**Chun Jason Xue**, City University of Hong Kong;

**Jihong Kim**, Seoul National University

### 背景与问题

本文针对移动闪存设备的文件碎片化问题进行了系统研究，文章对于智能手机文件碎片实验观察发现两个现象：①手机中文件碎片问题频繁出现并且严重影响性能；②闪存设备的文件碎片化问题与磁盘大不相同，其碎片化问题可分为逻辑碎片化与物理碎片化，逻辑碎片化问题较为常见并且影响较大，但是两种碎片化问题是解耦的。逻辑碎片化与传统文件碎片化问题相同，连续的文件数据块逻辑地址不连续，本文的物理碎片化问题定义为：逻辑连续的数据没有存放在并行的物理单元（channel、chip等）上，从而影响设备并行性，降低系统性能。传统的文件碎片整理工具如e4defrag需要进行大量数据复制操作，降低闪存寿命故不适用于闪存设备，需要提出新的去文件碎片工具。

### 方案设计

本文针对闪存设备特有的解耦的逻辑/物理碎片化问题，提出了新的文件碎片整理工具janus defragger (janusd)，其包含两个解耦的碎片整理模块：①JanusdL 针对文件逻辑碎片化问题，修改闪存设备的映射表，将不连续的文件地址重映射到连续的地址；②JanusdP 针对存储在同一并行单元的连续数据进行迁移（涉及数据复制，只在有需要的情况下执行）。其整体结构如图1.1 所示，DoFL和DoFP分别代表逻辑碎片化成都和物理碎片化程度，分别在文件系统和闪存设备获取获取，然后调用FTL的碎片整理工具。

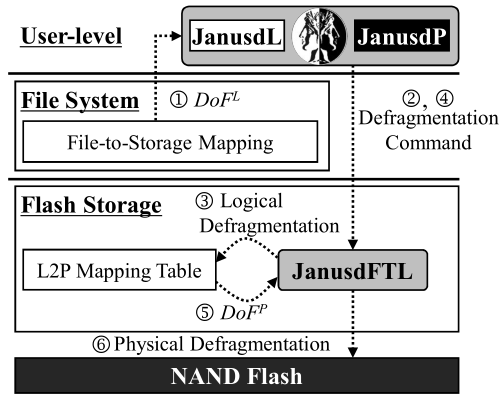


图1.1 janusd整体结构

其中逻辑碎片整理（JanusdL）原理如图1.2所示。

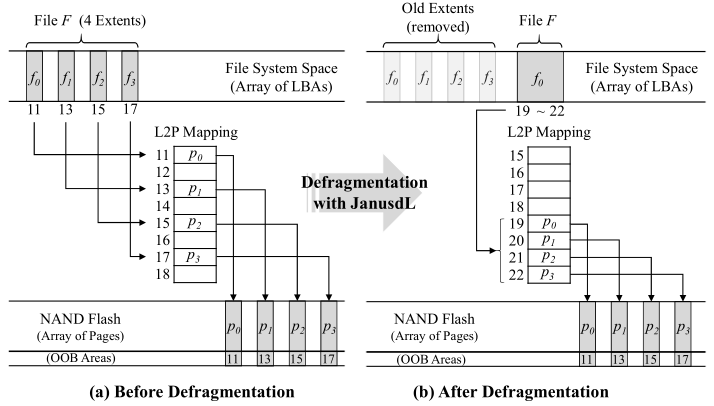


图1.2 JanusdL重映射

物理碎片化整理涉及数据复制迁移，只在DoFP低于阈值才会被调用，避免产生过多的数据迁移。

通过这种解耦的碎片化整理工具，针对闪存设备特殊的碎片化问题，提升移动闪存设备的性能。

### 实验及结果

**实验方法与设置：**使用定制的SSD加上SATA扩展命令实现仿真器simeMMC、simUFS，使用trace收集器收集真实手机产生的系统调用作为输入。仿真平台如图1.3所示：

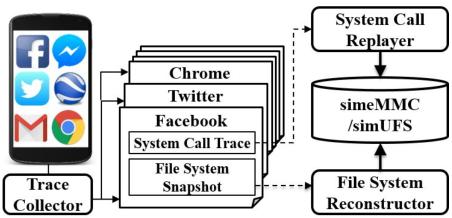


图1.3 实验平台框架

**实验结果：**图1.4显示了模拟六个月使用情况的归一化吞吐率，其中e4defrag\_nw表示每n个week执行一次e4defrag，可以看到使用文章提出的janusd和e4defrag\_1w取得的效果相同，但是使用e4defrag会引起大量数据复制，降低闪存寿命，而本文的janusd产生的数据复制较少。

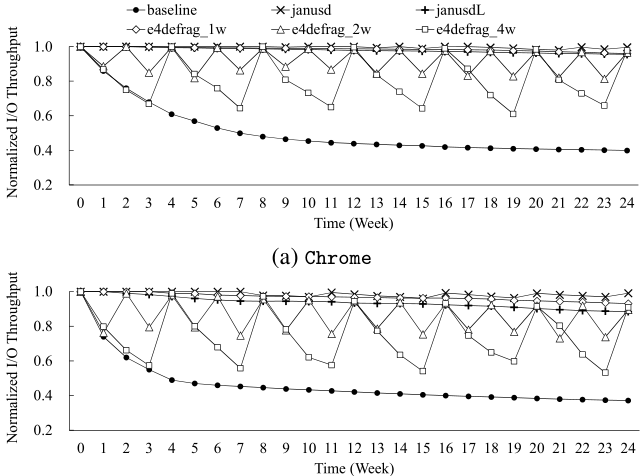


图1.4 实验模拟六个月手机使用情况

## FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones

### 出处与作者

出处：USENIX ATC ’18

作者：**Sangwook Shane Hahn**, Seoul National University;

**Sungjin Lee**, DGIST;

**Inhyuk Yee**, AIBrain Asia;

**Donguk Ryu**, Samsung Electronics;

**Jihong Kim**, Seoul National University

### 背景与问题

智能手机的用户体验很大程度上取决于一个APP对于用户操作的响应速度。尽管现在智能手机对于前台应用和后台应用的activities做了很多区分，但是在本文实验中发现前后台应用在IO路径上的处理优先级没有进行区分，尤其是在page cache和闪存设备上，后台IO较多，严重影响了前台应用IO响应时延。

文章经过仿真实验发现后台IO（更新、备份、下载等）对前台IO（Android系统Activity栈的栈顶activity IO）的影响集中在三个部分：

①page cache页分配锁争用：当只有一个前台应用产生IO需要page cache分配page时，可以立即获取page cache锁，但是如果存在后台IO，后台IO分配页占用page cache锁，阻塞前台IO，增大时延。具体如图2.1所示。

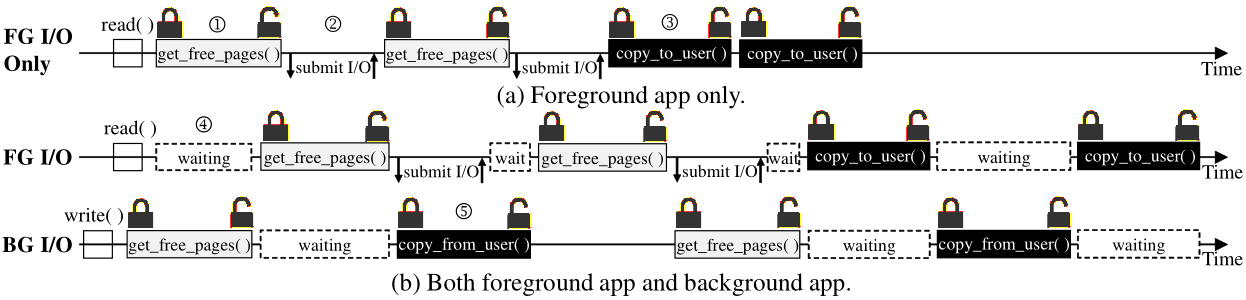


图2.1 page cache页分配

②page cache页替换：当后台应用需要page cache分配大量空间时，kernel可能会将前台应用的page剔除，当前台应用再次读取该页时，需要从闪存督导page cache，导致延时增加。具体如图2.2所示。

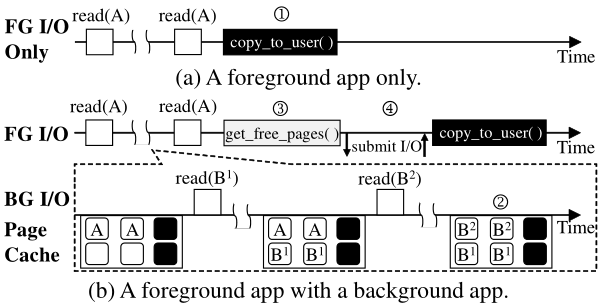


图2.2 page cache页替换

③设备内部IO调度：闪存设备内部自带的IO调度对前台IO不感知，影响IO执行顺序。

### 方案设计

针对以上问题，本文提出了针对前台应用IO优化的方案—FastTrack。其在Android系统中获取Activity栈顶activity的UID，并获取该activity的所有进程PID，建立两者联系。将前台activity进程产生的IO进行标记并在IO路径各个环节给与高优先级：①page cache页分配时优先为前台IO分配页；②page cache页替换时优先替换后台应用的page，保证前台应用page访问；③IO请求下发时适当优先下发前台应用IO；④IO调度适当优先前台IO先服务。整体设计如图2.3所示。本文整体设计完全优先于Android前台应用IO，可能在某些情况下（前台应用响应需要后台IO支持等）效果不明显甚至降低性能，需要仔细分析。

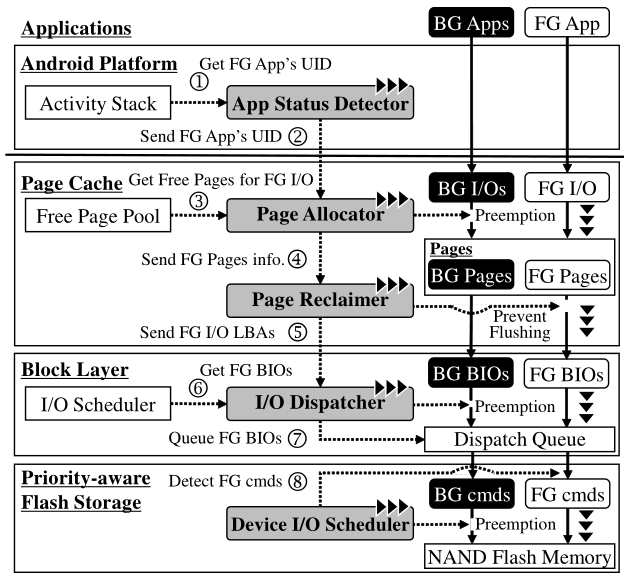


图2.3 FastTrack整体框架

### 实验及结果

**实验方法与设置：**文章作者在Android 7.1.1和linux 3.10.61中实现了FastT，并且在4个手机上进行实验。使用Update和Backup分别作为写为主和读为主的后台应用，然后测试4个手机应用启动的响应时间。

**实验结果：**结果如图2.4所示，其中PA、PR、ID分别代表仅优化IO路径中的一部分，PA代表page allocator，PR代表page reclaimer，ID代表I/O dispatcher。可以看到在仅有前台应用的情况下响应时间相对很短，如果存在后台应用则响应时间变长，使用了FastT其中的某一个优化或者使用整个FastT都可以降低相应时间，证明提出方案的有效性。

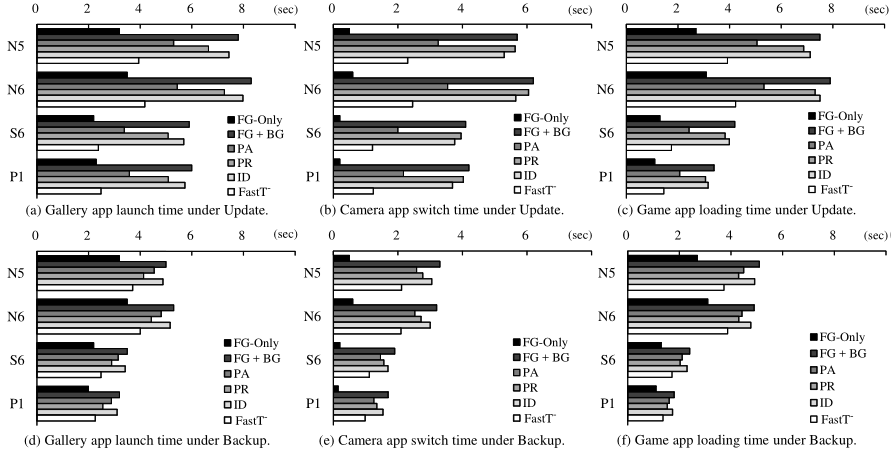


图2.4 应用响应时间对比

## I/O Stack Optimization for Smartphones

### 出处与作者

出处：USENIX ATC ’13

作者：**Sooman Jeong**, Hanyang University, Seoul, Korea;

**Kisung Lee**, Samsung Electronics, Suwon, Korea;

**Seongjin Lee**, Hanyang University, Seoul, Korea;

**Seoungbum Son**, Samsung Electronics, Suwon, Korea;

**Youjip Won**, Hanyang University, Seoul, Korea;

### 背景与问题

本文针对智能手机的I/O栈进行了研究。安卓系统的I/O栈由DBMS、文件系统、块设备驱动及存储设备组成，是应用程序用于管理数据持久性的一组软硬件。大多数安卓应用利用SQLite管理数据持久性，SQLite通过通用系统调用与底层文件系统交互并以日志形式进行恢复。数据库文件与日志文件通过fsync()调用与存储设备频繁的进行同步。文章对EXT4、BTRFS、XFSD、NILFS和F2FS五种文件系统进行了性能分析并着重研究了文件系统与各模式SQLite日志的交互方式，观察到：通过fsync()调用，文件系统重复记录了数据库的日志行为且频繁对SQLite的数据库文件和日志文件进行提交，将这一现象称为JOJ（Journaling of Journal）。

### 方案设计

本文针对智能手机I/O栈中的JOJ现象提出了以下三个改善性能的方案：

①利用fdatasync()代替fsync()以减少文件系统中不必要的元数据日志：fsync()调用同时将元数据和数据刷新至存储设备；fdatasync()调用仅在需要正确处理后续数据检索是同步元数据。利用fdatasync()调用代替fsync()调用能够缓解JOJ现象带来的性能下降。

②选用最适宜的SQL日志模式；

③外部日志：EXT4和XFS可以将日志块存储于外部块设备，通过这一方法可以降低随机性并利用空间局部性聚类写操作，从而提高性能。

④基于轮询的I/O：利用轮询I/O避免多核CPU及低延迟环境下由上下文切换造成的性能下降。

### 实验及结果

①利用fdatasync()代替fsync()以减少文件系统中不必要的元数据日志方案的实验验证

**实验方法与设置：**对EXT4、BTRFS、XFS、NILFS和F2FS五种文件系统进行调用替换，于数据库包含1000个条目的16GB Transcend SD卡上测试SQLite中insert与update操作的性能变化。X轴中B表示baseline，即使用fsync()的文件系统；F表示使用fdatasync()的文件系统；E表示外部日志。

**实验结果：**如图1.1所示。对于插入操作，该调用替换方案使得EXT4实现了17%的性能提升；F2FS改善最多，改善后插入速率比EXT4的插入速率快111%；BTRFS和NILFS2几乎不变（插入操作导致新块的分配，新块中元数据均需同步）。对于更新操作，EXT4和XFS改善效果明显，分别提高了50%及66%（更新操作相当于对已有数据库记录的覆盖写，并未改变元数据）；BTRFS和NILFS2几乎不变（属写时复制文件系统，更新操作导致新块的分配和后续元数据的更新）；F2FS改善最多，提高了250%。综上，将fsync()调用替换为fdatasyn()调用可以在一定程度上提高性能。

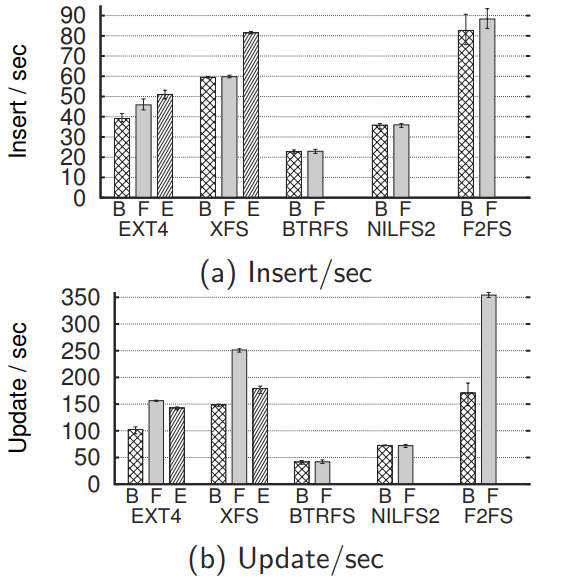


图1.1 调用替换方案实验验证结果

②使用最适宜的SQL日志模式方案的实验验证

**实验方法与设置：**分别对EXT4、BTRFS、XFS、NILFS和F2FS五种文件系统测试五种SQL日志模式下（DELETE、TRUNCATE、PERSIST、WAL及OFF）的性能（insert与update操作）。SQL的五种日志模式分别对应图中D、T、P、W、O五个缩写。

**实验结果：**如图1.2所示。选用WAL（Write-Ahead-Logging）日志模式可以获得较高的性能（所有数据库操作产生的日志数据均存放于.db-wal文件中且仅产生一个fsync()调用），F2FS是性能最佳的文件系统（fsync()调用仅产生两个分别用于数据和元数据同步的随机I/O，远远少于其他文件系统）。综上，选用F2FS文件系统及WAL日志模式可获得较高的性能。

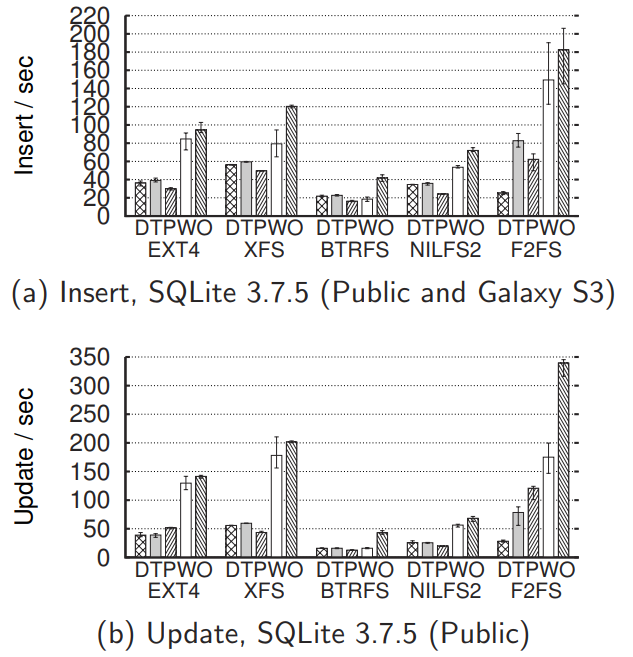


图1.2 调用替换方案实验验证结果

③外部日志方案的实验验证

**实验方法与设置：**同实验①，其中，16GB Transcend SD卡作为外部日志存储日志文件，eMMC存储数据。

**实验结果：**如图1.1所示。使用外部日志方案后，对于插入操作和更新操作，EXT4及XFS两种文件系统的性能均有所改善。

④基于轮询的I/O方案的实验验证

**实验方法与设置：**

**实验1（图1.3）：**分别测量中断驱动I/O(baseline)与轮询I/O情况下的（非自发、自发及全部）上下文切换次数；

**实验2（表1.1）：**在轮询I/O子系统上分别运行1个线程和10个线程，其中各线程均会因fsync()调用产生4KB大小的随机写。

**实验结果：**

**实验1：**如图1.3所示。采用轮询I/O后，自发上下文切换次数减少至原来的1/100，全部的上下文切换次数减少为原来的1/50；

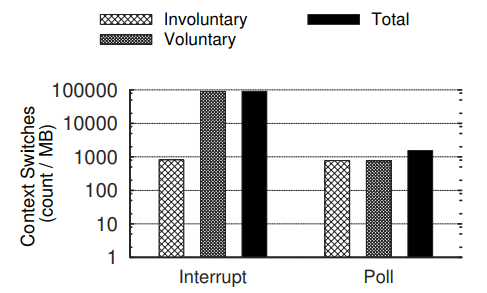
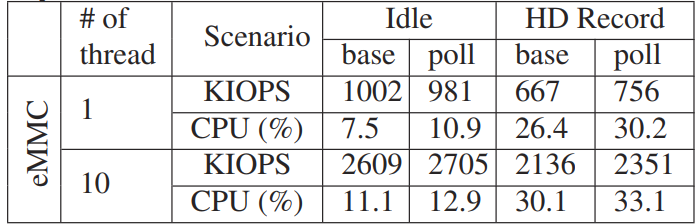


图1.3 中断驱动I/O(baseline)与轮询I/O的上下文切换次数

**实验2：**如表1.1所示。采用轮询I/O后，性能（观察KIOPS值）有所提高。

表1.1 fsync()调用产生的4KB随机写的吞吐量（eMMC/EXT4）



⑤投入实际进行实验

**实验方法与设置：**从常用应用推特和脸书上抓取trace并利用Mobigen回放，分别于五种文件系统上对不同优化方案测试执行时间。X轴上B表示baseline，即使用fsync()的文件系统；P表示使用轮询I/O；F表示使用fdatasync()的文件系统；E表示外部日志。

**实验结果：**如图1.4所示。类似实验①②，各方案均对性能有一定改善，且五种文件系统中F2FS效果最佳。

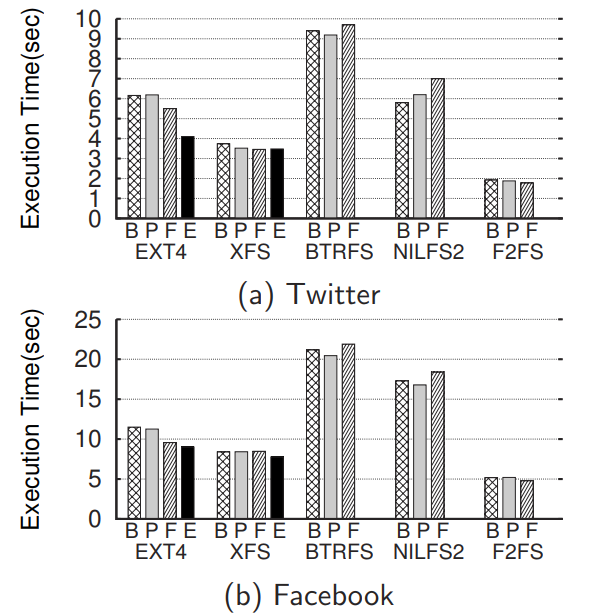


图1.4 回放测量执行时间

⑥综合优化方案的实验验证

**实验方法与设置：**

**实验1（图1.5）：**对EXT4（baseline）、XFS和F2FS三种文件系统综合实施上述三种优化方案（调用替换、外部日志及轮询I/O），SQLite日志模式默认选用TRUNCATE，测量各种情况下的SQLite性能。

**实验2（表1.2）：**对EXT4（baseline）、XFS和F2FS三种文件系统综合实施上述三种优化方案（调用替换、外部日志及轮询I/O），SQLite日志模式选用最适宜的WAL，测量各种情况下的SQLite性能。

**实验结果：**

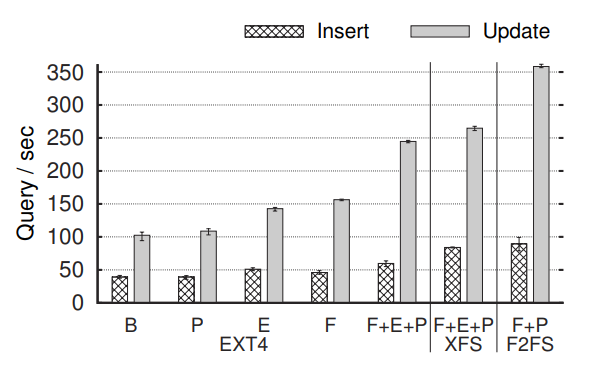
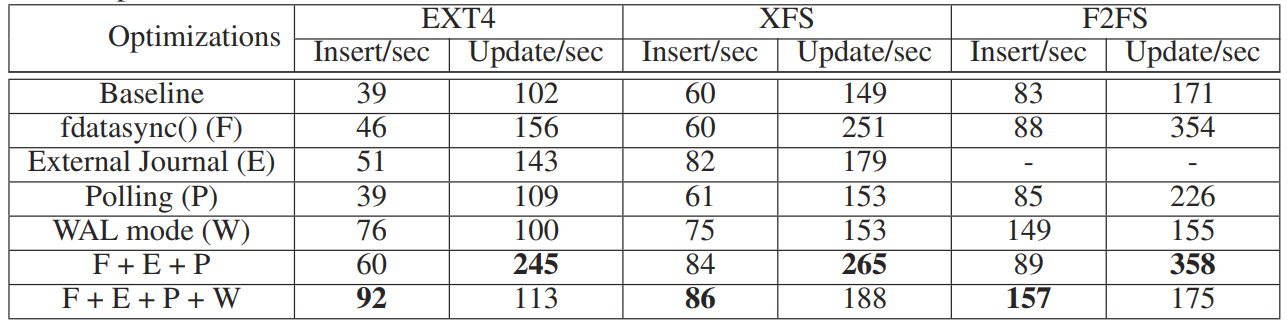


图1.5 综合优化方案的实验验证

**实验1：**如图1.5所示。对于EXT4，综合实现上述三种优化方案，插入操作性能及更新操作性能分别提高了53%和130%；对XFS综合实现上述三种优化方案则将收获更高的性能；对于F2FS，综合实现fdatasync()和轮询I/O，插入操作性能及更新操作性能分别提高了130%和250%，效果最佳。

**实验2：**如表1.2所示，性能均有所提高。

表1.2 综合优化方案的实验验证



## Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices

### 出处与作者

出处：FAST’15

作者：Daeho Jeong, Samsung Electronics Co.;

Youngjae Lee and Jin-Soo Kim, Sungkyunkwan University

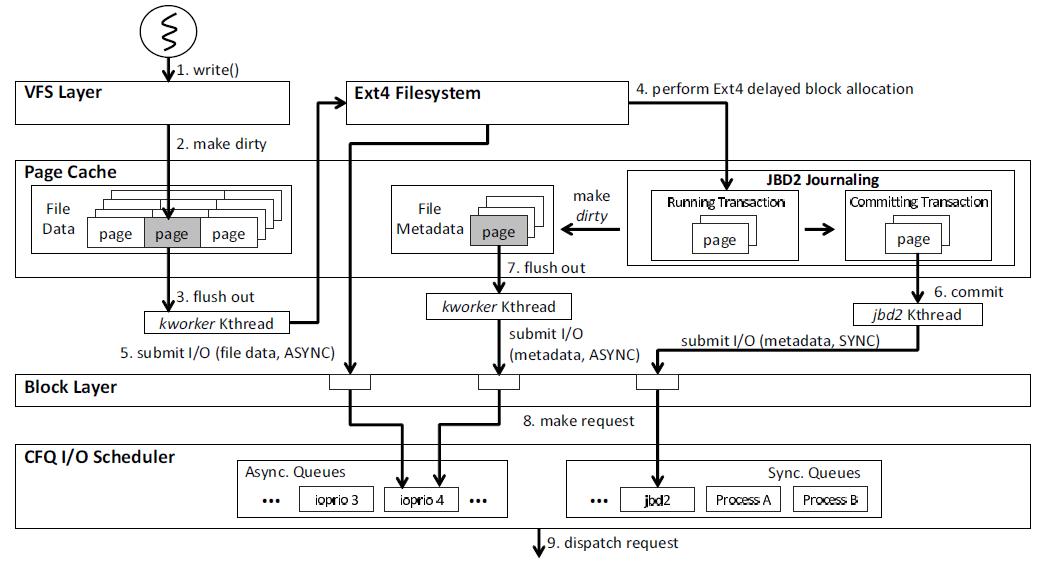
### 背景与动机

移动设备越来越成为消费者最流行的电子产品，在使用消费级的电子产品时，体验永远是用户最看重的一点，手机或者平板电脑的流畅度最能带来直观的感受。目前手机等消费产品中使用的存储空间是越来越大，硬件配置像CPU主频和核数也越来越高，但是存储的性能与响应的时间延时一直都是在逐渐优化过程。

这篇文章中主要集中文件系统的几个常用的接口操作：create()，write()，truncate()，fsync()等。作者通过观察发现，在Android系统中比较大的负载情况下，这些文件系统的操作是对应用的IO响应影响是比较大，特别是前台应用在运行过程中，后台存在大量的异步（asynchronous）IO，而且作者发现其实就是前台的应用在某一个时刻等待异步IO的完成，这与之前认识的异步IO特点就是不需要等待互相矛盾。

文章引入了一种新的IO操作类型，Quasi-Asynchronous I/O（QA-SIO），主要特点是本身以异步下发的IO操作，但是应该被当做同步IO处理，原因是有其他的任务在等待其完成。异步IO提升为QASIO是在运行时完成。

文章给出了基本动机之后，首先给出了文件系统操作的基本方式，以文件的写操作为例，给出了内核中在不同的存储层次上IO的调用路径，具体如下图所示：



文章把IO分为了如上的9个步骤，步骤1是用户层面的写请求操作，此时会从用户态陷入内核态，产生写操作的系统调用，进入VFS层。VFS层会更新文件系统必要的一些元数据（时间戳，日志信息等），同时将请求的数据写入到页级缓存中。步骤2是将页级缓存中指定的页置为脏页，脏页在特定的情境下会被刷到存储介质上。步骤3是linux按照时间和脏页比例的场景，将脏页以相应频率刷下出缓存。刷新的数据进入文件系统层，步骤4是文件系统对执行的数据块的分配，注意文件系统的底层存储单元仍然是以块为单位。步骤5是进行脏页的写回，此时产生的异步的IO请求。步骤6是对于元数据的修改，linux是先写日志，在去修改元数据持久化，所以步骤6即是写日志操作产生的是同步IO请求。操作完成之后，在去更新元数据，并执行步骤7，即产生的异步的IO操作。步骤8则是块设备层进行产生块级IO请求。步骤9则是将不同类型的请求放置到不同的IO队列中，由IO调度器进行统一的IO调度。

在分析了内核的IO栈操作之后，作者给出了在实际应用场景中观察到前台应用发生较大的延时的几种现象，分别是：

* Launching CONTACTS APP

当后台4GB的文件正在进行拷贝的时候，进行打开某些CONTACTS应用，整个打开时间增加了140%

* Burst Mode in the CAMERA APP

连拍情况下，如果前一次连拍结束，每次拍摄的照片大小设置8MB，后面继续连拍就会出现两次之间的延迟，整体性能大概下降了19%

* Installing the ANGRY BIRDS APP

当后台4GB的文件正在的拷贝的时候，安装整个应用平均时间增加了35%

作者发现上述问题有一个共同的地方是，存在一系列的文件系统的操作被一些异步的IO阻塞，导致前台的任务等待后台的异步IO的完成，从而导致比较大的延迟。这也进一步说明了作者提到的准异步IO对于前台应用的阻塞问题。

### 问题分析

作者在发现了上述的问题后，重新对问题进行了深入的分析，作者首先分析了准异步IO（QA-SIO）的产生与哪些系统调用有关，并列举了4类强相关。

* 当更改元数据页的时候；这种类型主要是这样：当一个任务调用文件系统操作更改元数据页时，目标元数据可能被它自己或者其他任务dirty，可能已经被kworker作为异步IO下发。
* 当要更改数据页时；当一个任务要去在数据页中进行追加等，如果这时目标页已经被kworker以异步IO下刷，这时就需要进行等待。
* 当保证数据要被writeback时；如果一个进程调用fsync()或者truncate(), 当要指向fsync的时候，所有在pagecache中的buffer数据都需要同步下刷，但是如果fsync调用时，已经有部分的dirty page异步下刷，这时就会出现fsync要等待所有的数据页都变成最新才返回，就会有等待异步IO完成的情况。
* 当要完成discard命令时；当前的系统中，jbd2内核线程是异步发送discard命令，其他的journal则是同步方式；因此，它的执行将会在每次journal提交的时候被阻塞，一直到所有的discard命令都执行完毕。

以上的四类被作者认为是强相关的，即和准异步IO（QA-SIO）直接相关，此类IO会直接导致应用的系统调用产生较大的延时。另外一种是间接阻塞，即任务的执行另外一个任务被QASIOs阻塞，作者也给出了两种间接型阻塞的特性。

* 当不能获取到journal句柄，在Ext4文件系统中，任务在修改元数据页或者数据页时首先要获取journal句柄。前面已经提到，如果目标页已经异步写入，这时就会出现阻塞。随后包含journal句柄的事务需要提交，但是这个事务会进入locked状态，因为阻塞的任务正持有journal句柄，这时候另外要进行文件操作的任务因为获取不到新的journal句柄而被阻塞。
* 由于discard不能完成fsync()；fsync系统调用正常情况下需要等待journal提交完成，保证对应文件的元数据持久化，然而，journal提交的处理时间可能因为jbd2内核线程异步发送discard命令而被加长。

在分析了与哪些系统调用有关的过程后，作者再次从应用下手，分析了上述三种场景下

如何产生准异步IO（QA-SIO）导致应用IO延时的。

* 场景A：启动CONTACTS APP

当一个app的UI要在Android上进行显示的时候需要文件操作更新到数据库SQLite中，启动CONTACTS需要一系列系统调用，如：rename()，write()，fsync(), unlink(). 这些系统调用需要更新元数据，因此他们在遇到大量异步IO的情况下会产生元数据类型的QASIOs

另外就是UI任务在每次journal提交时可能因为jbd2内核线程执行discard命令产生delay。

* 场景B：连拍模式

连拍模式下，因为thumbnail maker任务重复写入1KB的小数据，可能造成目标数据页已经writeback而产生延迟。

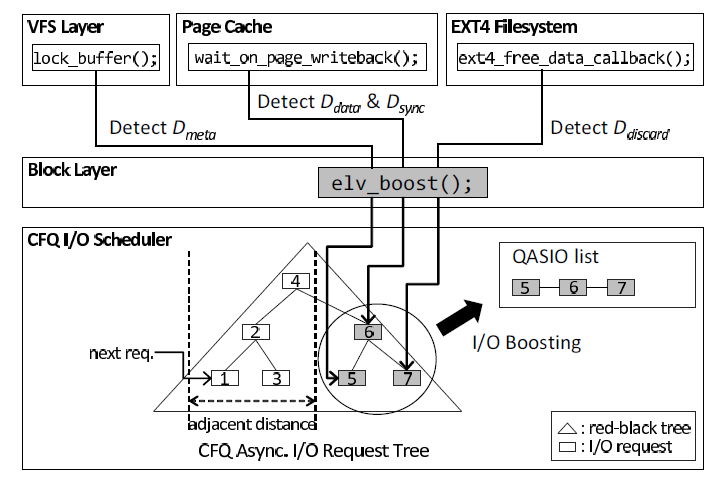
另外，thumbnail maker任务在每次写入数据页的时候都需要获取journal句柄，如果由于上面的原因sleep，经过一定周期时间jbd2提交就会进入locked状态，这时CAMERA想要获取journal句柄进行文件修改就会被阻塞。

* 场景C：安装ANGRY BIRD APP

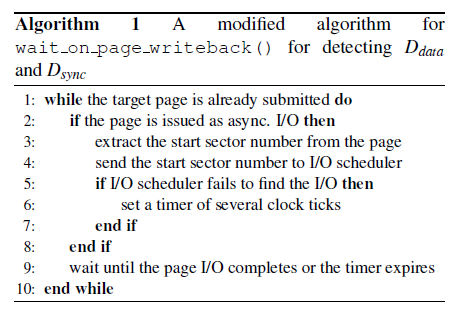
安装APP需要将大量数据写入到底层设备中，并会调用fsync及时下刷，但是如果在后台有大文件的拷贝，page cache就会充满dirty page，这时如果准备被fsync刷入的页前面经过kworker异步发送，就会导致同步情况的QASIOs

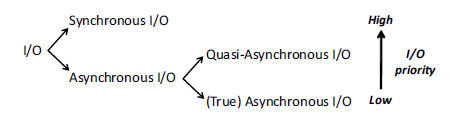
### 方案设计

作者发现了准异步IO对于前台应用的阻塞问题，提供了一种解决此问题的方案。该方案通过在线实时监测哪些IO是QASIO，将QASIO的IO优先级进行提升，放在单独的IO队列上，让QASIO能更快的执行，从而减小阻塞的时间。系统方案的整体设计图如下图所示：



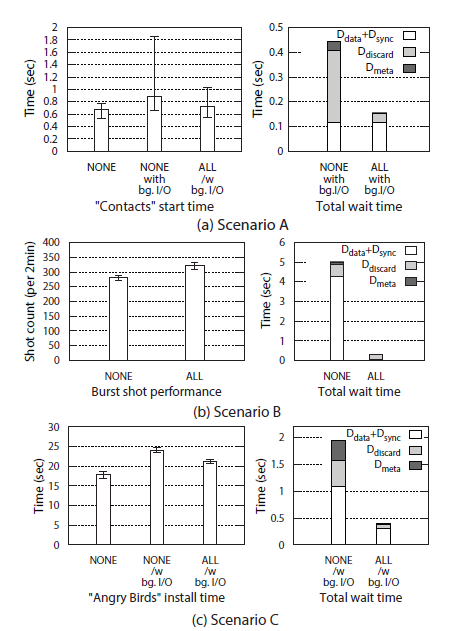
具体来说，方案分为两步，第一步是检测哪些是QASIO，第二步是修改IO调度器调度策略，提高QASIO的优先级，让其先执行。其中，检测QASIO的算法如下：



优先级提升的方法，即将QASIO的优先级提升高于异步IO，但是低于同步IO。

### 实验结果

文章最后通过在真实的三星手机上进行上述方案的实现，并测试了在上述三种场景下延时的提升，相对于传统，该方案能有效的降低应用操作延时，比例达到了87.4%~96.1%。三种场景下的结果土如下图所示：



# 一般IO栈优化

## Enlightening the I/O Path: A Holistic Approach for Application Performance

### 出处与作者

出处：FAST ’17

作者：**Sangwook Kim**, Apposha and Sungkyunkwan University;

**Hwanju Kim**, Sungkyunkwan University and Dell EMC;

**Joonwon Lee** and **Jinkyu Jeong**, Sungkyunkwan University

### 背景与问题

在数据密集型的应用，如数据库或KV存储中，一个用户请求的响应时间决定了用户可感知的性能，因此，将前台任务（用户请求等）与后台任务（IO密集的内部任务，如checkpoint、backup等）进行IO优先级划分是十分重要的。但是现有的IO优先级静态划分策略：①按照IO类型划分，如同步IO高优先级；②按照任务类型划分，如前台IO高优先级，存在两个问题：①没有系统考虑IO栈的多层独立结构，cache、FS、block、device等，只在一个或两个层次保证优先级无法满足要求；②没有解决任务或IO运行时的依赖关系导致的IO优先级倒置的根本问题，如图3.1所示：

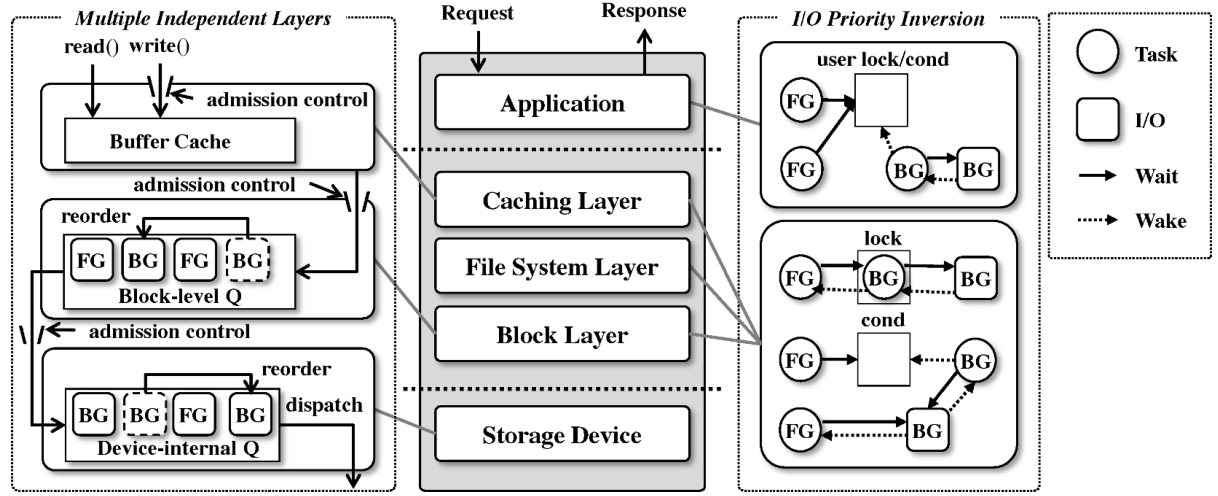


图3.1 OS存储栈多层独立结构

* 1. Caching layer：系统所有任务都以dirty page ratio控制写回，可能由后台任务造成的大量脏页阻塞了前台任务的IO响应；
  2. FS layer：文件系统将要写回的数据混合放到一个事务中提交，没有考虑IO的优先级；
  3. Block layer：块层队列容量有限，如果突发后台IO占据大量空间，前台IO响应无法保证；
  4. Device internal：设备内部会对IO重新排序，也会影响IO的执行顺序。

运行时依赖也会引起IO优先级倒置：①任务依赖，两个任务通过同步原语（锁、条件变量等）相互影响；②IO依赖：任务需要等待某个IO完成以保证数据正确性和/或持久性。现有的方法解决上述依赖关系存在挑战：①依赖关系在运行时可以动态变化而不是静态的；②依赖存在传递性，涉及并发任务多个阶段、多个层次；③现在许多基于内核支持的用户级同步功能（如共享内存互斥等）引起的依赖关系在内核层不可见。

### 方案设计

文章将IO类型分为两类：①关键IO，位于用户请求处理关键路径上的IO；②非关键IO，并非位于关键路径的IO。区别于传统的优先级静态划分策略，如图3.2所示。基于这种以请求为中心的IO优先级划分，将IO路径上所有层次的关键IO与非关键IO分离。

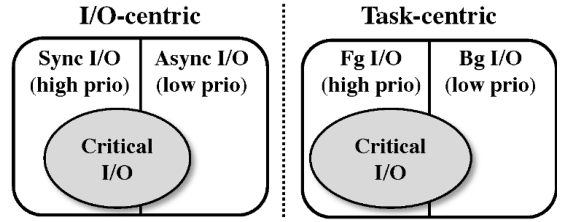


图3.2 IO优先级划分方法比较

判断关键IO，第一步要判断task是否为前台应用，有两种方法：①在内核层使用可用信息判断；②由应用提供准确信息。本文采取方法②，提出set/clear接口来标记/清除前台任务标记，供应用的某段程序显示调用。判断出前台任务之后，先简单的把前台任务的同步IO判断为关键IO，第二步要解决运行时依赖造成的IO路径各层优先级倒置，本文提出了IO优先级继承策略：给予前台任务依赖的后台任务或后台IO 临时关键IO优先级。具体实现：①解决任务依赖：当后台任务拥有的锁阻塞了前台任务，给予后台任务高优先级直到它释放锁；②解决IO依赖：追踪非关键IO处理过程中的状态，若其阻塞了前台任务，将非关键IO调整为关键IO优先级。

在IO路径各层次对关键IO和非关键IO执行不同的管理策略：①caching layer：设置两个相互独立的dirty page ratio；②FS和block layer：分配独立的请求提交IO；③scheduler：设置两个FIFO队列，关键IO先分派，设置定时器防止非关键IO饿死；④device layer：适当限制非关键IO提交给device的速率。通过在IO路径各个层次判断IO类型，保证请求处理关键路径的IO优先完成，提升请求响应。

### 实验及结果

**实验方法与设置：**Dell PowerEdge R530服务器，两个Intel Xeon E5-2620处理器和64 GB RAM，一个1TB Micron MX200 SSD，使用Ubuntu14.04以及3.13版本Linux内核和ext4文件系统，使用CFQ调度器作为时延基线，CFQ-IDLE表示提升前台IO优先级（将后台IO优先级设置为IDLE），还使用Split-AFQ (SPLIT-A) 、Split-Deadline (SPLIT-D), 和QASIO等调度器作为对比。对于本文的RCP为关键IO配置回写脏页比率为20%（linux内核默认比例），非关键IO脏页回写比例为1%。下发非关键IO到设备的数量设置为1，超时时间设置为10ms防止饿死。

**实验结果：**①使用OLTP-Bench为PostgreSQL生成TPC-C工作负载。模拟了50台客户机在一台单独的机器上运行30分钟。PostgreSQL配置为初始数据集大小的40％缓冲池并且每30秒checkpoint一次。对于CFQ-IDLE和QASIO，将checkpointer置于空闲优先级。对于SPLIT-A，分别为backends和checkpointer设置最高和最低I / O优先级。对于SPLIT-D，分别为backends和checkpointer设置5 ms和200 ms fsync（）截止时间。图3.2显示为事务响应延时互补累积分布，纵坐标表示大于某一数值的概率，可以看到本文提出的RCP方案明显减小时延。

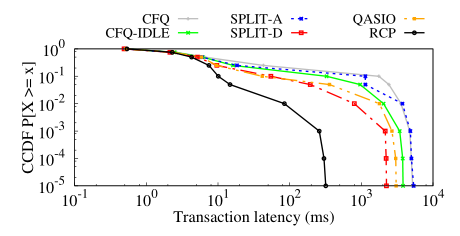


图3.3 响应延时分布

②对于MongoDB，在YCSB基准测试套件中使用了更新繁重的工作负载（工作负载A），模拟150台客户在一台单独的机器上运行30分钟。MongoDB配置了大小为初始数据集40％的内部缓存。图3.4显示其相应延时互补累积分布，可以看到本文RCP方案延时小于其他方案。

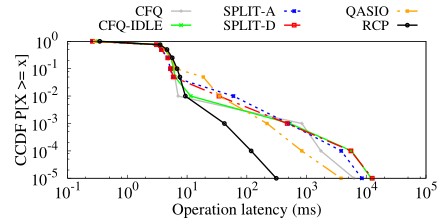


图3.4 响应延时分布

## Unblinding the OS to Optimize User-Perceived Flash SSD Latency

### 出处与作者

出处：HotStorage ’16

作者：**Woong Shin**, Seoul National University

**Jaehyun Park**, Arizona State University

**Heon Y. Yeom**, Seoul National University

### 背景与问题

现在高性能的闪存SSD可以将随机访问延时控制在微秒级别，但是消耗在IO路径软件上时间占比增大，导致用户感知的性能没有达到SSD的极限。一个重要的问题是SSD请求响应时间浮动较大，OS对SSD内部请求响应无法感知，使得任务在等待请求完成时难以决定是释放CPU还是阻塞CPU以等待请求完成，因为任务调度产生的时延不可忽略，如果单纯阻塞CPU等待请求完成，又会影响系统并行性，特别是现在SSD内部多并行，增加任务调度的情况发生的频率。具体如图4.1所示。

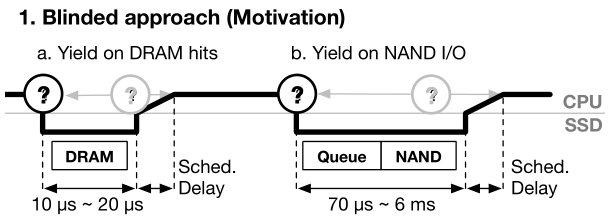


图4.1 调度延时

* 1. 如图4.1 a所示，请求在SSD内部DRAM上响应，响应时间较短，原本不需要进行任务调度，但是下发该请求的task已经放弃CPU使用，请求完成发出中断之后需要进行上下文切换，产生sched delay；
  2. 如图4.1 b所示，请求在SSD的NAND芯片上响应，下发请求的task放弃CPU使用，在请求完成发出中断之后再进行上下文切换，总时延包括请求响应时延加上 sched delay。

以上两种情况都是OS无法判断SSD内部请求处理的情况，造成sched delay影响用户体验。

### 方案设计

根据以上观察，本文提出一种SSD与OS协同优化的方案，目的在于保证系统并行性的同时，消除或隐藏任务调度时延（sched delay）。具体实现如下，在SSD内部增加IO行为追踪器：①监测SSD内部资源（DRAM、NAND闪存芯片）的状态；②追踪每一个请求的完成状态。主机OS增加IO延时预测器：①帮助OS预测IO请求的响应延时；②请求完成前中断内核。具体如图4.2所示：

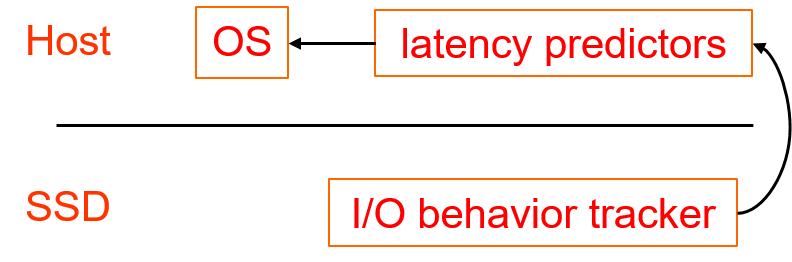


图4.2 整体架构

IO行为追踪器收集SSD内部资源以及IO完成状态信息，上报给Host端的延时预测器，延时预测器判断某个IO请求是否会在SSD内DRAM响应，从而告知OS是否可以直接阻塞CPU等待响应，可以消除sched delay；或者对于某些可预测完成（文章中提到的size<8KB的Read）请求，在其完成之前一段时间内通知OS，可以提前进行任务调度，可以隐藏sched delay。具体如图4.3所示：

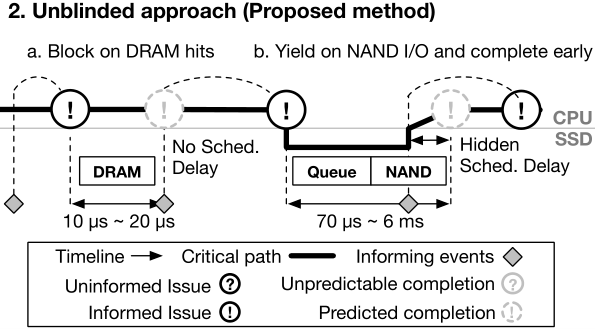


图4.3 预测请求完成

本文的方案中可预测的IO范围较小，仅包括DRAM响应的请求以及size<8KB的Read请求，主要由于主机端排队时延、SSD内部DRAM flush以及SSD内部排队时延无法准确预测，需要大量的信息及准确的模型才能预测。

### 实验及结果

**实验方法与设置：**在OpenSSD2 Cosmos验证平台上的“Greedy FTL firmware”实现了本方案SSD原型，使用PCI-Express Gen1 x4连接主机，主机内核在Linux 3.5.0上进行相应修改，由于该测试板只有一个IO context，实验使用Fio 2.1.3 IO作为资源竞争者，使用另一个消耗CPU的程序作为CPU资源竞争者，为了解预测请求完成的优点，文章实验展示了后台任务去除NAND延时之后的平均延时以及吞吐量，并与后台任务单独执行情况进行了对比。

**实验结果：**

Precompletion I/O vs I/O Threads：图4.4中的浅灰色柱子和线条表示后台任务与I / O线程（Fio）竞争的情况。轮询使用（POLL \_PRIO）或不使用（POLL）任务优先级（niceness）显示最佳延迟，而中断（IRQ）表示调度延迟高达11μs。减少这种调度延迟的成本是CPU周期过多导致后台I / O线程吞吐量降低17.75％。

预测完成通过屏蔽SSD I / O时间下的调度延迟（11μs），同时不损害后台线程，解决了延迟和并行之间的这种困境。但是，应根据对调度延迟的观察给出精确的预完成窗口。在图4.4中，使用16μs预完成窗口（PRE16）实现了最佳延迟。

Precompletion I/O vs CPU Threads：图4.4中的深灰色条纹和线条显示了后台任务与CPU线程相互竞争的情况。测得的POLL的延迟明显更大（1,488μs），而后台吞吐率没有显着下降。相反，基于中断的I / O线程（POLL和POLL\_PRIO以外的线程）没有遇到此问题。

POLL\_PRIO缩短延迟的成本是后台CPU任务性能的显着下降，与单独运行场景相比仅有80％的吞吐量（ops / sec）。然而，预完成有效地减少了这种延迟，而不会严重降低CPU任务。

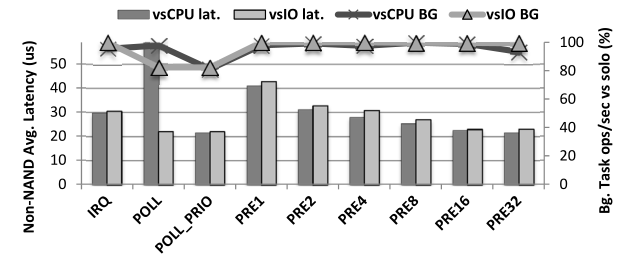


图4.4 实验结果对比

## Barrier-Enabled IO Stack for Flash Storage

### 出处与作者

出处：FAST ’18（Awarded Best Paper!）

作者：**Youjip Won**, Hanyang University;

**Jaemin Jung**, Texas A&M University;

**Gyeongyeol Choi**, **Joontaek Oh**, and **Seongbae Son**, Hanyang University;

**Jooyoung Hwang** and **Sangyeun Cho**, Samsung Electronics

### 背景与问题

现代的Linux IO协议栈是每一层都会依照自己的方式处理IO。由于多层的复合不确定性，保证数据的存储顺序写入存取器的开销十分大。通常利用“转移并刷新”(*Transfer-and-Flush*)机制来保证前面的请求相关联的数据块完全转移到存储设备并使其持久之后才调度后续的请求。但是存在两个关键问题：①闪存的性能和容量在不断的提升，内部的并发性和并行性也显著提高，而基于*Transfer-and-Flush*的顺序保持机制与闪存存储中的并行性和并发性相冲突，开销巨大。在有32通道闪存阵列的SSD中有序写入的性能是无序写入的1%。②现有的保证顺序存储的工作（OptFS、Featherstitch等）都依赖于*Transfer-and-Flush*机制。

对于一个闪存设备来说，一个写请求真正的写入存储设备需要一个复杂的路径。依次是主机端的IO调度、IO分发、传输至闪存存储设备以及最后的从cache中写入flash。如图5.1所示。

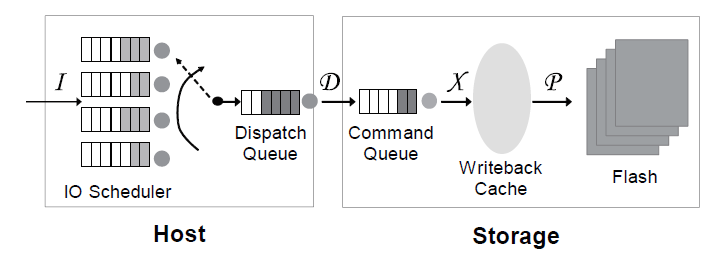


图5.1 IO栈

### 方案设计

整体设计在块设备层和文件系统层均有涉及。

保序块设备层由基于epoch的IO调度器、保序分发模块和新定义的barrier write命令组成。保序分发模块消除了*Wait-on-Transfer开*销，新定义的barrier write命令消除了*Wait-on-Flush*开销。它们共同工作来保证存储顺序，而不需要*Transfer-and-Flush*机制。保存顺序块设备层将写请求分为两类，无序写和保序写。无序写在一个epoch可以任意重新排序。带REQ\_BARRIER属性的保序barrier写用于分隔epoch。其他保序写则带有REQ\_ORDERED属性。图5.2为Barrier-Enabled IO栈的组织示意图。

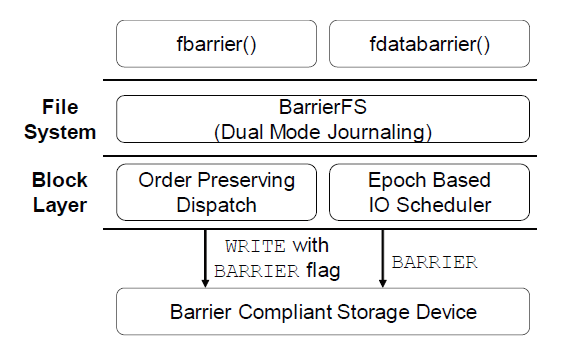


图5.2 barrier-enabled IO栈

在文件系统层，通过修改EXT4文件系统来实现启用barrier的IO栈，不同与Linux原本的fsync()、fdatasync()同步原语，本文提出了两个新的文件系统接口fbarrier()和fdatabarrier()来支持顺序保证。fbarrier()和fdatabarrier()分别使用fsync()和fdatasync()同步同一组块，但是它们返回时没有确保关联的块变得持久。fdatabarrier()是一个通用存储屏障。通过将write()调用与fdatabarrier()交叉，就能确保fdatabarrier()之前的写请求的数据一定在fdatabarrier()之后的写请求数据块执行之前变得持久。简单的说fdatabarrier()的作用类似与一个“挡板”，保证了其前后命令的顺序。

除此之外该文章实现了Dual Mode Journaling（双模式日志记录机制），利用双线程大大提升了事务处理的速度。双模式日志记录机制支持两种日志记录模式，持久性保证模式和顺序保证模式。以及提交多个事务，冲突页面可能与不同的提交事务相关联的multi-transaction page conflict情况，利用了新的一个结构conflict-page list来解决Dual Mode Journaling带来的缺点。

### 实验及结果

**平台**：分别在企业服务器(12核，Linux 3.10.61)、PC服务器(4核，Linux 3.10.61)和智能手机(Galaxy S6, Android 5.0.2, Linux 3.10)三个不同的平台上实现了一个支持barrier的IO栈。

**存储设备**：分别对应三种存储设备:843TN (SATA 3.0, 队列深度=32, 8通道,超级电容)，850PRO (SATA 3.0, 队列深度=32, 8通道)和移动存储(UFS 2.0, 队列深度=16，单通道)。分别称为为supercap-SSD、plain-SSD和UFS。

并且假设在plain-SSD和supercap SSD中barrier write的性能开销分别为为5%，0。

**负载测试**：利用含大量fsync()的两个工作负载：varmail和OLTPinsert测试持久性和顺序性。在plain-SSD中，在varmail工作负载中，BFS-DR比EXT4-DR带来60%的性能提升。在supercap-SSD中，BFS-DR比EXT4-DR带来10%的性能提升。在排序保证方面，BarrierFS对EXT4-OD的性能提高了80%。与基线EXT4-DR相比，当我们在纯SSD中仅使用BFS-OD执行排序保证时，BarrierFS的性能达到了36 倍。在MySQL中，BFS-OD占EXT4-OD的12%。与基线EXT4-DR相比，当我们在plain-SSD中仅使用BFS-OD执行排序保证时，BarrierFS的性能达到了43倍。

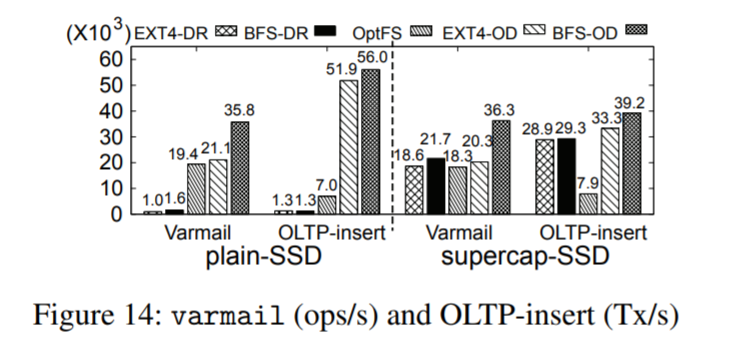


图5.3 varmail和OLTPinsert测试

针对移动存储，在持久化和WAL两种日志模式进行了测试。对于持久性保证模式，在移动存储中BarrierFS在缺省持久化日志模式下比EXT4的性能提高了75%。在顺序性保证模式中，在plain-SSD中，SQLite在BFS-OD中对EXT4-DR的性能提高了73倍。

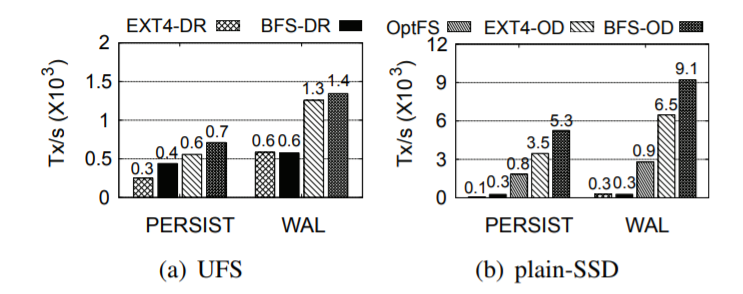


图5.4 两种日志模式测试

# 数据中心存储栈优化

## Split-level IO Scheduling

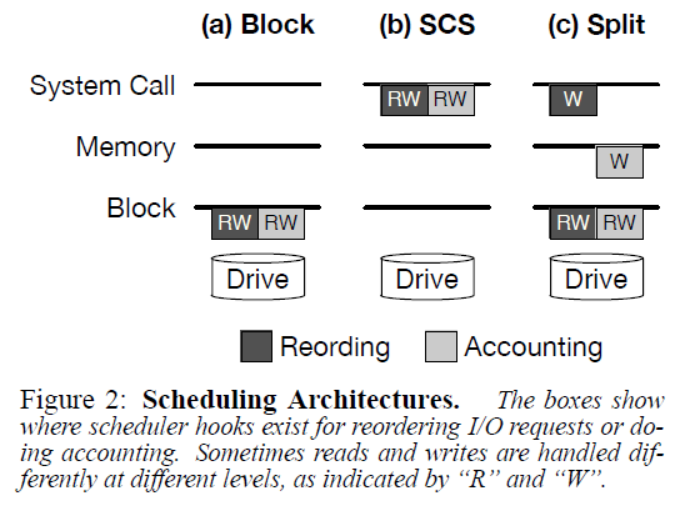
### 出处与作者

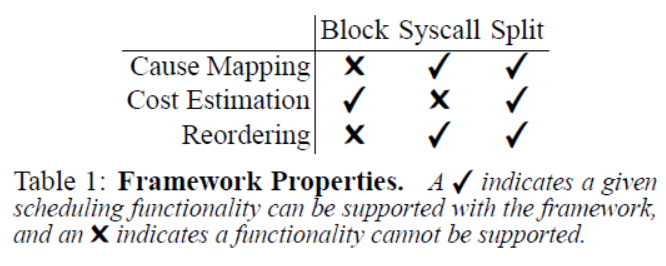
出处：SOSP’15

作者：Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik\*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. University of Wisconsin-Madison, IBM Research-Almaden\*

### 背景与问题

IO调度器的目标在于提高系统性能和提供QoS特性（比如优先级、截止时间和资源控制），为此它需要实现三个关键功能：(1) cause mapping，即IO溯源，识别下发的IO请求来自于哪一个进程；(2) cost estimation，即开销评估，据此进行性能优化和计算资源消耗；(3) reordering，即重排序IO请求以实现多样的调度目标。





大部分IO 调度器工作在操作系统的块层，处于文件系统和块设备之间，面临着两个问题：(1) 块层的调度器在重排序写请求上存在根本性的限制，因为文件系统会控制写顺序以保证一致性；(2) 在块层，调度器不能够精确地计算应用的IO开销，因为缺乏必要的上层信息导致无法判断有的IO请求来自于哪一个进程。

相比而言，工作在系统调用层的调度器避免了这两个问题，但是由于缺乏下层的信息，无法精确地评估一个系统调用带来的IO开销。

这些问题的产生主要来源于文件系统的设计。(1) Delayed writeback and allocation. 延迟的写回机制和地址分配机制会产生代理进程，即由一个代理进程代替其它进程来完成IO工作，比如 writeback process代替其它进程下发数据写请求，并产生元数据写请求，这导致块层的调度器无法判断是哪一个进程产生的脏数据。(2) Journaling. 日志技术给IO溯源带来了挑战，因为内核进程会代理写文件的进程下发相应的事务写和检查点操作；而且，阻碍了IO请求的重排序，因为文件系统的一致性机制会对写入顺序进行限制，比如日志合并（不同进程的写请求导致的日志写被合并为一个日志操作）使得一个写请求的完成依赖于其它IO请求的完成。(3) Caching and write amplification. 文件系统的数据布局（影响了IO的顺序性）、数据缓存以及造成的IO放大（比如元数据IO和日志写）使得系统调用层的调度器无法精确地评估一个系统调用的IO开销。

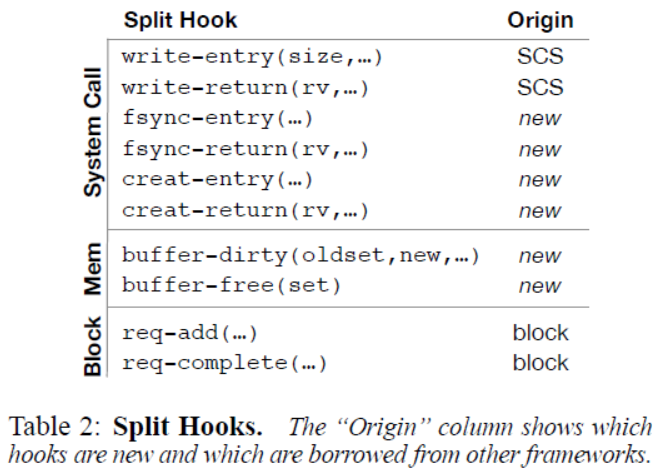
### 方案设计

单层的调度框架存在各自的限制，因此本文提出一个分离的调度框架（split framework），在IO栈的多个层次上将处理IO的关键决策暴露为调度钩子（scheduling hooks），然后利用这些钩子实现以下三个关键功能。

* **IO溯源（Cause Mapping）：**分离框架给IO请求打标签，记录它的来源集合；给代理进程打标签，记录它所服务的进程集合。
* **开销评估（Cost Estimation）：**由于写请求会被缓存，因此在块层进行开销评估时，在快速评估和准确评估之间存在一个权衡。分离框架在缓存层和块层暴露调度钩子，使得调度器能够根据调度目标在快速和评估之间取得权衡。
* **重排序（Reordering）：**在块层暴露钩子，允许重排序读请求和写请求；在系统调用层暴露钩子，允许重排序写系统调用、fsync系统调用以及其它会修改元数据的系统调用（比如mkdir, creat）。

基于Linux实现分离调度器：

* 跨层的标签：给写请求添加一个 causes tagging 数据结构，设置写回进程和日志进程为代理进程；
* 在系统调用层、缓存层和块层三个层次上设置调度钩子。



### 实验结果

实例探究：

* **AFQ: Actually Fair Queuing.** 在块层调度读请求，在系统调用层调度写系统调用以及会产生写请求的系统调用（比如fsync），从而根据优先级实现公平调度。相比于CFQ，AFQ将优先级偏离程度降低了28倍。
* **Deadline.** 利用缓存层的脏数据信息评估fsync一个文件的开销，然后针对开销大的fsync操作，提前下发异步写减少脏数据，避免了fsync操作带来大开销的突然性IO，从而降低尾延时。相比于Block-Deadline, Spit-Deadline将尾延时降低了4倍。
* **Token Bucket.** Split-Token在系统调用层限制写，在块层限制读，并利用缓存层和块层的钩子计算IO开销。相比于SCS（系统调用层的调度器），Split-Token将进程间干扰降低了6倍。
* **运行真实的应用：**在数据库、虚拟机和分布式文件系统上，分离调度能实现更好的性能和隔离。

## FLASHSHARE: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs

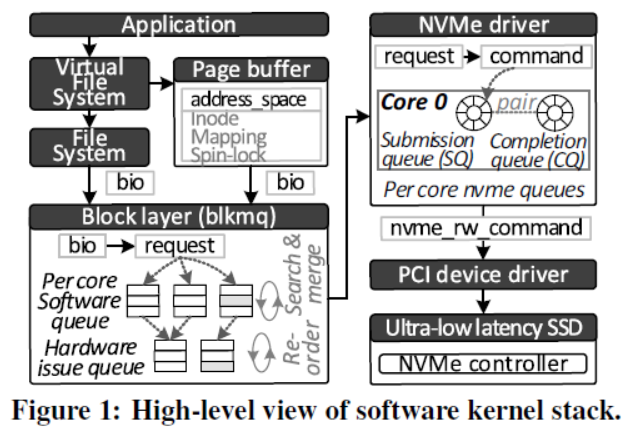
### 出处与作者

出处：OSDI’18

作者：Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Yonsei University, University of Illinois Urbana-Champaign, Seoul National University, Pennsylvania State University.

### 背景与问题

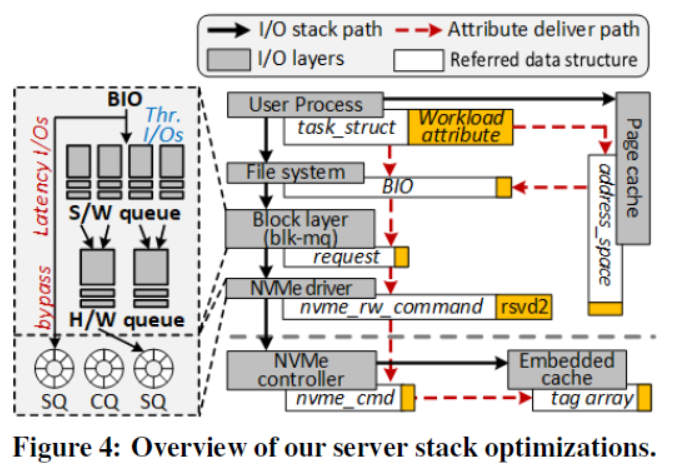
数据中心的服务器上通常会同时运行多个应用，包括延时敏感的应用。简单地将存储设备替换为低延时SSD（比如Z-NAND, 3D-Xpoint）并不能满足应用低时延的要求，因为多个应用的IO请求处理会互相干扰，而服务器的存储栈不了解IO请求的关键程度，不能隔离干扰。



### 方案设计

本文提出一种整体的跨层的设计FlashShare，基于低延时SSD满足不同应用对IO延时的不同要求。FlashShare跨越整个存储栈，优化IO请求的下发、执行和完成。

* **内核排队优化：**多队列块层（block-mq）和NVMe协议层对请求进行排队，会带来较大的延时。FlashShare扩展内核数据结构，允许用户为应用动态地配置负载属性的信息，以区分延时敏感的应用（在线应用）和吞吐量导向的应用（离线应用），并将这些属性沿存储栈从内核向设备固件传递。针对延时敏感的应用，为了降低内核中的排队延时，FlashShare将它们的IO请求绕过多队列块层，并在NVMe驱动层为延时敏感的应用和吞吐量导向的应用分别设置独立的提交队列，并为前者设置更高的优先级。
* **IO完成优化：**基于轮询（poll）的IO完成机制比默认的基于MSI中断的IO完成机制具有更短的延时（平均12%），以及更高的CPU开销。FlashShare使用轮询机制处理延时敏感的应用的IO完成，而依然使用中断机制处理吞吐量导向的应用的IO完成。
* **SSD固件优化：**SSD内部通常配置有大容量的DRAM用来缓存数据，吞吐量导向的应用的IO请求可能会抢占大量DRAM资源。FlashShare将DRAM分为独立的两个缓存，分别供两类应用使用，并根据两类应用的IO请求数量动态调整两个缓存的大小，利用ghost caches动态调整预取粒度。
* **IO栈加速器：**通过PCIe接口增加了一个IO栈加速器，用于管理blk-mq和NVMe的队列（查找和合并）以及处理轮询。



### 实验结果

FlashShare能够在多应用同时运行的环境下将平均延时和第99百分位的尾延时降低22%和31%。

## Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems

### 出处与作者

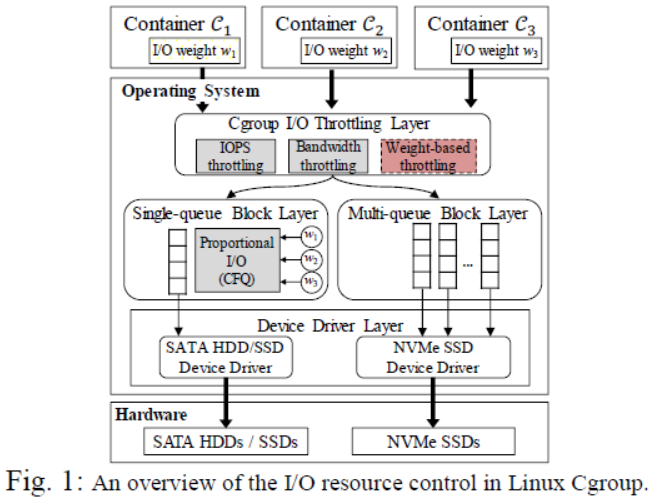
出处：USENIX HotStorage ’16

作者：**Sungyong Ahn** and **Kwanghyun La**, Samsung

**Jihong Kim**, Seoul National University

### 背景与问题

在基于容器的虚拟化环境中，在多个容器之间实现内核资源的隔离和共享是非常重要的系统需求。Linux Cgroup 提供了一个资源控制的框架，能够在Cgroup IO 节流层或者块层实现对 IO 资源的控制。例如，在基于单队列的块层中，利用CFQ 调度器里实现成比例的IO共享（proportional IO sharing）；在IO节流层中，限制一个容器的最大带宽或者IOPS（读、写各自独立限制）。但是，为了充分发挥出 NVMe SSDs 的高性能，块层采用了新提出的多队列机制，而现有的Linux Cgroup无法在多队列机制下提供成比例的IO共享。



### 方案设计

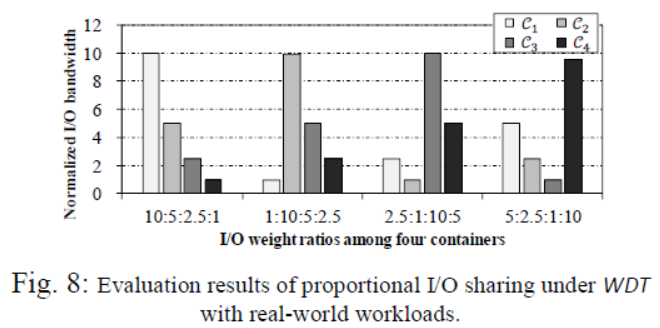
本文提出在Cgroup IO节流层实现基于权重的动态节流机制WDT，从而在基于容器的虚拟化环境下（在基于NVMe SSDs的NUMA多核系统中）实现成比例的IO共享。这里的IO共享是指locally-proportional IO sharing，即在给定的一个时段里，在有IO请求的多个容器之间根据权重成比例的分配IO资源。

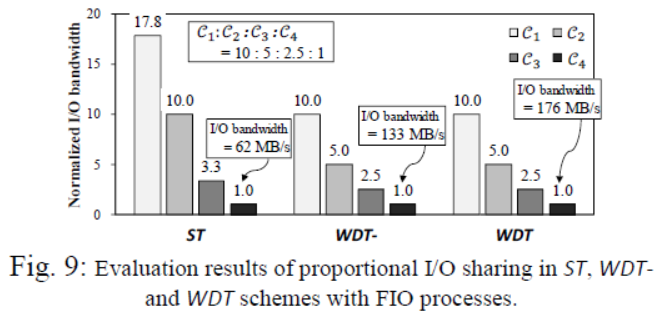
WDT在每一个固定长度的时间窗口里对容器进行IO节流，限制它们能够读写的数据量。WDT监测每个容器的IO到达速率，并基于历史速率预测未来的IO需求。周期性地（N个节流窗口），WDT计算下一个节流窗口的IO需求总和，然后将等量的IO预算根据权重分配给各个容器，并允许容器将当前节流周期未使用完的预算累积到下一个节流周期中。

另外，在实现WDT的过程中，发现Cgroup IO节流层的实现存在扩展性问题，即在检测IO带宽时，所有的容器共享一个请求队列的锁，导致IO性能在NUMA节点数增加时下降。因此，WDT在Cgroup IO节流层为每一个容器设置独立的锁。

### 实验结果

WDT实现了成比例的IO共享，并且相比于静态限制容器最高IO性能的方法ST，提高了总体的IO性能。





ST：利用传统的Cgroup，限制每个容器的最大IO带宽；

WDT-：没有为每个容器设置独立的锁。

## IO Priority: To The Device and Beyond

### 出处与作者

出处：USENIX HotStorage ’17

作者：**Adam Manzanares** and **Filip Blagojevic** and **Cyril Guyot**, Western Digital Research

### 背景与问题

**概要：**在数据中心中，存储设备的 IO 调度器导致了不可预测的尾延时，本文提出将应用的优先级信息传递给存储设备的调度器，从而降低尾延时。

在大规模数据中心中，控制 IO 请求的尾延时能够提供限定的和可预测的存储性能，这对于规划基础架构的资源非常关键。存储设备的IO 调度器提高了 IO 性能，但是也导致了不可预测的尾延时，特别是对于在云存储中广泛部署的磁盘而言。

现有的两种解决方案：(1) 优先处理 real-time IO 请求（相比于 background IO 请求），其问题在于有效性，因为不能将主机端的优先级信息传递给存储设备，存储设备的调度器可能会违反优先级；(2) 限制存储设备的 IO 请求队列的长度（也就限制了设备 IO 调度器的影响），这是很多数据中心的默认做法，其问题在于降低了存储设备的性能。

动机实验表明，(1) 磁盘的吞吐量反比于 IO 请求延时，即 IO 请求深度越大，请求延时越高，吞吐量也越高；(2) 主机端的 IO 调度器中，CFQ 支持优先级特性，但是在 foreground work 的请求队列深度很低时，出现了与 background work 之间的优先级反转。

### 方案设计

本文提出将应用的优先级信息传递给存储设备的 IO 调度器，从而降低调度造成的尾延时。传递通道包括应用、Linux 块调度层以及设备的 IO 调度器，其实现利用了现有的 IO 优先级 API，并已经被合并进 Linux 4.10 内核。

具体实现：

* 1. **应用层：**应用利用 read()/write() system call 下发 IO 请求，并可以利用 iopriority\_set() 设置 IO 优先级；
  2. **VFS 层：**read/write requests 被转化为 bio structures；
  3. **Block 层：**bio structures 被转化为 request structures，IO 调度器会对它们进行请求重排序和合并，并可以从 iocontext structure 中获取优先级信息。**修改：将 iocontext 中的优先级信息与 request 联系起来**；
  4. **Device driver 层：**requests 被转化为接口命令（比如 SATA/NVMe），添加到存储设备的命令队列中。**修改：利用 request 优先级信息进行优先级调度。**

### 实验结果

消除了优先级反转的问题，增加了foreground work 的 IOPS。

