|  |  |
| --- | --- |
| 学号 | 2017202110053 |

**海量存储结课论文**

|  |
| --- |
| **在移动设备中提升准异步I/O以提高响应速度** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 计算机系统结构 |
| 学 生 姓 名 ： | 张琴 |

二〇一七年十二月

摘 要

由于移动设备的交互性，快速的系统响应对其非常重要。然而，我们观察到，当后台有大量异步I/O操作时，文件系统的操作延迟急剧增加。仔细分析得知，大部分的延迟是由意外情况导致的。在这种情况下，文件系统操作被堵塞直到一个或多个I/O操作完成。由于多个任务被阻塞在一个地方，这种异步I/O就具有了同步I/O一样的属性。我们把这种异步I/O叫做准异步I/O，Quasi-Asynchronous I/O (QASIO)。

我们将任务与QASIO之间的依赖关系进行分类，然后显示在Linux内核中这些依赖关系何时出现。除此之外，我们提出了一种新的方案，用以在运行时检测QASIO并且通过I/O调度器中的其他异步I/O来提升它们。我们在最新的智能手机上的测试结果表明，这种方案能够有效提高文件系统操作的响应能力。

目 录

[1 简介 5](#_Toc32043)

[2 背景 7](#_Toc27922)

[2.1 Andriod平台的I/O 7](#_Toc10422)

[2.2 Linux内核I/O路径 7](#_Toc17266)

[3 问题和动机 10](#_Toc31544)

[4 准异步I/O(Quasi-Asynchronous I/O，QASIO) 12](#_Toc10536)

[4.1 准异步I/O的定义 12](#_Toc6142)

[4.2 QASIO的依赖关系类型 13](#_Toc14396)

[4.3 回顾问题场景 14](#_Toc19766)

[4.3.1 场景A：启动CONTACTS应用 15](#_Toc14680)

[4.3.2 场景B：CAMERA应用的连拍模式 15](#_Toc3889)

[4.3.3 场景C：安装应用ANGRY BIRDS 16](#_Toc11416)

[5 提升QASIO 17](#_Toc28288)

[5.1 设计 17](#_Toc13728)

[5.2 检测QASIO 18](#_Toc26662)

[5.3 提升QASIO优先级 20](#_Toc28053)

[6 评估 21](#_Toc10295)

[6.1 方法论 21](#_Toc29102)

[6.2 微基准测试 22](#_Toc23200)

[6.3 现实场景 23](#_Toc31606)

[6.4 Android的I / O基准测试 25](#_Toc17125)

[7 相关工作 27](#_Toc30710)

[8 结论 28](#_Toc23320)

[参考文献 29](#_Toc32098)

# 简介

智能手机和平板电脑等移动设备已经成为最受消费者欢迎的电子设备之一。 据Gartner调查估计，2013年全球移动设备出货量为20亿台，预计到2015年将比传统PC高9倍[7]。

就移动设备而言，快速响应能力是提供令人满意的用户体验的关键。因为运行在移动设备上的大多数应用程序需要一直与用户进行交互，所以快速而没有任何明显延迟的系统响应非常重要。虽然存储容量在增加且其性能有明显的提高，但是存储仍然经常要为损害移动设备终端用户体验负责[9]。

研究人员们已经做了很多工作来了解流行移动应用的I / O特性及其对底层存储介质NAND闪存的影响[9,12]。 基于这些研究，人们针对包括文件系统，I / O调度器和块层在内的移动平台I / O栈提出了各种优化[10,8,11,18]。 然而，以前的方法主要是从增加吞吐量或延长NAND闪存寿命的角度来看待问题的。

在本文中，我们重点关注在繁重的I/O负载下，诸如creat()，write()，truncate()，fsync()等文件系统操作的延迟。我们观察到，当系统发出大量异步I/O请求时，这些文件系统操作延迟会急剧增加，从而应用程序的响应能力严重下降。在我们对一款最新的基于Android的智能手机的评估中，当大量文件写入在后台进行的情况下，应用程序的启动时间已经放慢到2.4倍。

未来随着移动设备的外围设备不断采用更新，更先进的技术，这个问题将会变得更加严重。 例如，最新的智能手机配备了Wi-Fi 802.11ac（1Gbps）和USB v2.0（480Mbps）模块，可以产生几十兆字节/秒的I / O流量。 用户很可能通过Wi-Fi或USB连接在后台下载一些大文件的同时运行应用程序。 在这种情况下，前台任务的响应将受到大量异步I / O操作的显着影响。

当前台任务在大量异步I / O的情况下访问文件时，只要共享相同的存储设备，一定程度的延迟是不可避免的。然而令人惊讶的是，我们发现大部分的延迟来自意外的情况，在这种情况下文件系统操作被无意地阻塞，直到一个或多个异步I / O操作完成。这种现象与传统的观点相矛盾，即在没有等待的情况下，可以随时执行异步I / O操作。因为异步I / O优先级比同步I / O要低，并且异步I / O处理的优化不是针对延迟而是针对吞吐量，所以当前台任务需要等待异步I/O操作完成时，其响应能力会受到很大影响。我们使用高端智能手机进行测试的结果表明，单独调用看似良性的creat（）或缓冲write（）系统调用时，当其执行过程被挂起的异步I / O阻塞，则可能需要1秒以上（参见表2）。 对于存储性能较差的低端智能手机来说，这些系统调用的最坏情况下的延迟通常会增加到几秒钟。

为了解决这个问题，我们引入了一种称为准异步I / O（QASIO）的新的I / O操作类型。 QASIO是一种异步发出的，但由于其被其他任务阻塞，所以应该被视为同步I / O进行处理的I / O操作。需要注意的是，当一个任务被一些异步I / O阻塞，这些异步I/O就在运行时成为QASIO。因为被阻塞任务的执行依赖于QASIO，QASIO应优先于（真正的）异步I / O以获得更好的响应性。 据我们所知，这项工作是第一个讨论文件系统操作和异步I / O操作之间依赖性的研究。

我们提出了一个新的方案来检测QASIO并在Linux内核中提升它们。 首先，通过对包含虚拟文件系统（VFS），页面缓存，Ext4文件系统，JBD2日志，I / O调度器和块层在内的Linux内核的整个存储I / O堆栈的全面研究，我们分析了三个由于QASIO而导致应用程序的响应性严重下降的有问题的情况。 然后，我们对文件系统操作和QASIO之间的直接或间接依赖关系进行了分类。 我们还介绍了如何检测Linux内核中的每种依赖关系。 最后，我们设计了一种机制在CFQ I / O调度程序中动态地调整QASIO的优先级。CFQ I / O调度程序是Linux内核中事实上的I / O调度程序。

我们已经在一款最新的基于Android的智能手机Samsung Galaxy S5上实现并评估了这个方案。我们使用微基准测试的评估结果显示，creat()，fsync()和缓冲write()的最坏情况延迟分别减少了98.4％，87.1％和90.2％。在实际工作负载中， 在繁重异步I / O下，CONTACTS应用的启动时间减少了44.8％。

本文的其余部分安排如下。第2部分解释了一些了解Linux内核如何处理文件I / O操作的背景内容。 第3节描述了三种有问题的场景，第4节介绍了QASIO。 第5节介绍了在Linux内核中检测和提升QASIO的方案的设计和实现细节。第6节展示了评估结果，第7章讨论了相关的工作。 最后，第8节总结全文。

# 背景

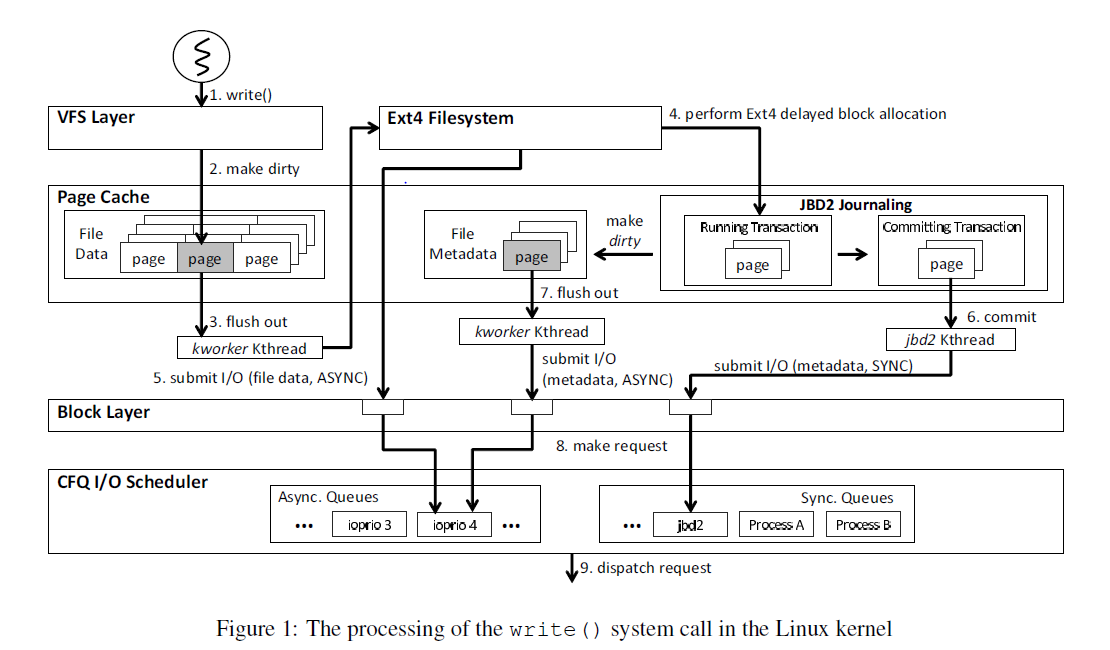
## Andriod平台的I/O

Android是世界上使用最广泛的移动平台之一。 Android提供了一种允许大量应用程序同时运行的应用程序框架。Android应用程序由不同类型的组件组成，如Activity，Service，Broadcast Receiver和Content Provider，它们具有各自的入口点，需要被作为独立任务来执行。 一个应用程序不仅提供用户界面，还提供各种后台服务。这些后台服务由应用程序的其他组件的请求执行，甚至可以由其他应用通过Intents [2]执行。 最近，Android设备开始支持多窗口模式[16]，而不是简单的多任务处理。多窗口模式允许用户将一个显示屏分成两个屏幕，并在每个屏幕上执行不同的任务。 由于这些原因，在Android系统中任何时刻可以同时运行大量任务，同时产生大量I / O。

## Linux内核I/O路径

Linux内核是Android平台的核心，负责管理CPU，内存，各种I / O设备等系统资源。 特别是存储I / O堆栈是Linux内核中最复杂的部分之一，因为每个文件系统操作都是借助于各种层，如虚拟文件系统（VFS），Ext4文件系统，页面缓存，JBD2 日志层，I / O调度器和块层来处理的。 在本小节中，我们将简要描述一个处理write()系统调用的分步过程，如图1所示。我们选择write()系统调用是因为，与其他系统调用相比由于要同时处理数据和元数据，它要稍微复杂一些。我们假设在Ext4文件系统中使用默认的有序日志模式。

1. 调用一个write()系统调用：任务通过write()将文件描述符，用户数据缓冲区的地址和写入大小传递给内核。 这个信息被传送到VFS层。 VFS通过获得JBD2日志句柄来更新文件元数据（即inode）的必需字段（诸如时间戳和文件大小）。 每次修改元数据以获取日志区域中更新的元数据时，都会获取日志句柄。 然后VFS将请求的数据复制到页面缓存中的相应页面。 一旦将数据复制到页面缓存中，调用任务将从write()中返回。
2. 使文件数据的页面变脏：由VFS写入的文件数据的页面被标记为变脏。 Linux内核将这些脏页累积到一定的阈值。
3. 清除脏页面：如果脏页面停留的时间超过了有效时间，或者脏页面的数量超过了后台脏污比例，它们将被强制刷新。 kworker内核线程定期或同步执行以满足这些要求，并要求Ext4文件系统在必要时刷新脏页。



1. 执行Ext4延迟块分配：默认情况下，当页面缓存的文件数据被刷新时，Ext4执行块分配。 在块分配期间，Ext4需要修改块位图和组描述符等文件系统元数据。 由于这些元数据也应写入日志区域，于是Ext4获取正在运行的事务的日志句柄，并请求JBD2模块进行管理。 需要注意到Ext4会将缓冲区头与每个元数据页关联起来。
2. 提交文件数据的脏页面：块分配后，Ext4将脏数据页面作为异步I / O提交给块层。
3. 提交JBD2日志：在预定的时间到达或者jbd2内核线程收集了一定数量的缓冲区头之后，正在运行的事务被更改为提交事务。 属于提交事务的元数据页面需要由jbd2提交为同步I / O，因为它们必须快速写入存储设备才能保证文件系统的完整性。
4. 清除脏元数据：日志提交完成后，提交事务中包含的元数据页面会再次变为脏状态，从而启动对元数据页面的检查点设置。 最后，kworker将这些元数据作为异步I / O刷新。
5. 提出请求：提交的I / O请求中具有相同I / O属性的物理相邻页面通过块层合并为一个请求。 这些请求被发送给I / O调度器。
6. 调度请求：CFQ是Linux内核中的默认I / O调度器。 CFQ有分别用于同步I / O和异步I / O的队列。 从进程生成的同步I / O请求被输入到为每个进程提供的同步CFQ队列中。 另一方面，异步CFQ队列由具有相同I / O优先级的进程共享。 由于大多数异步I / O是由kworker（I / O优先级4）提交的，如图1所示，于是它们被放入同一个异步CFQ队列。 在调度请求时，CFQ首先选择CFQ队列，然后按照逻辑扇区号的顺序处理选定队列中的I / O请求。 需要注意的是所有同步队列的优先级都高于CFQ中的异步队列。

# 问题和动机

本节介绍了三种真实场景，作为显示在繁重异步I / O下响应性降低的激励示例。 这些场景的性能结果可以从我们的测试设备获得（详情参阅第6部分）。

场景A：启动CONTACTS应用

应用程序启动延迟是一个简单的显示移动设备的响应速度上午指标。 我们观察到，当同时在后台创建4GB文件时，CONTACTS应用程序的启动过程变慢，并且随着大标准差而变化。 经过500次重复测试得出，最糟糕状况下应用程序启动时间比没有后台I / O流量的正常情况下增加了140.0％。

场景B：CAMERA应用的连拍模式

CAMERA应用程序中的连拍模式是一种连续的高速拍摄模式。现在许多智能手机都支持这一功能[19]。 我们的测试设备三星Galaxy S5通过触摸拍摄图标并保持，最多可拍摄30张照片。 在第一次连拍所拍摄的图像被处理后不久，下一次连拍就可能发生。 当每个镜头的图像尺寸很大（在测试情况下为8MB）时，在第一次连拍之后的拍摄中会发生间歇性延迟。 最后，连拍性能比理想情况下降了19.0％。

场景C：安装应用ANGRY BIRDS

最后一种情况是在后台写入4GB文件时，安装从Google Android市场下载的ANGRY BIRDS应用程序。 由于安装包文件的下载取决于网络性能，于是我们将软件包文件预先下载到本地存储，然后测量安装应用程序的时间。 我们观察到当后台有异步I / O时，平均安装时间增加35.0％。

根本问题：

在安装或启动应用程序期间，会发出许多同步I / O，例如read()或write()以及fsync()。 如第2.2节所述，CFQ I / O调度程序为这些同步I / O提供了比异步I / O更高的优先级。 尽管如此，由于以下原因，前台任务在繁重异步I / O下有一些延迟是不可避免的。 首先，在锁定修改文件系统元数据时可能会出现争用。 其次，当同步I / O被调度时，存储设备中的另一个异步I / O可能已经在进行中。 第三，没有可用的内存空间或请求队列分配给额外的I / O。

然而，仔细研究这三个场景之后，我们发现前台任务下发的文件系统操作是由另外一个原因造成的。 虽然具体情况稍有不同，问题的根源是一样的。即文件系统操作进程被未分派的异步I / O阻塞。 这在Linux内核中是一个意料之外的情况，它导致了前台任务需要等待I / O调度程序中排队的异步I / O完成的不愉快结果。 由于这些异步I / O尚未分派，所以延迟可能会很长，因为I / O调度程序可能会最后才处理这些异步I/O。 在下一节中，我们将更详细地展示何时发生这种情况。

# 准异步I/O(Quasi-Asynchronous I/O，QASIO)

本节定义了一种称为准异步I / O（QASIO）的新的I / O类型，并描述了QASIO的依赖类型。 我们还回顾了第3部分中的问题场景以演示这些场景与QASIO的关系。

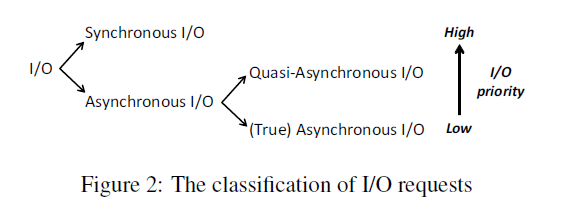
## 准异步I/O的定义

Linux内核传统上将I / O请求分为以下两类：

同步I / O：当调用任务一直被阻塞直到I/O请求完成，我们把这样的I / O请求称作是同步的。 出于这个原因，CFQ等I / O调度程序优先于异步I / O处理同步I / O以获得更好的响应性。 通常同步I / O由read()，fsync()和sync()系统调用创建。 但是，使用O SYNC标志打开文件可以使write()同步。 jbd2内核线程在提交日志数据时也会生成同步I / O。

异步I / O：将数据写入没有用O SYNC标志打开的文件将创建异步I / O。异步I / O由kworker线程一起刷新以最大化I / O吞吐量。 它们在I / O调度程序中处于低优先级，因为没有任务等待它们。 这样任务可以自由享受缓冲I / O的好处。 在文件系统元数据在日志提交之后被写入原始位置时，也会生成异步I / O。

在本文中，我们将介绍一种称为准异步I / O（QASIO）的新的I / O类型，如图2所示。QASIO被定义为看起来异步但具有同步属性的I / O， 因为有一个或更多的任务在等待它的完成。 这在理论上似乎是不可能的，但我们在下一小节会展示它在实践中经常发生。 请注意，I / O请求是同步的还是异步的是在提交给块层时确定的。 相比之下，当任务由于异步I / O而被阻塞时，现有的异步I / O就在运行时被提升为QASIO。 为了提高响应速度，应该赋予QASIO比其他（真正的）异步I / O更高的优先级。



## QASIO的依赖关系类型

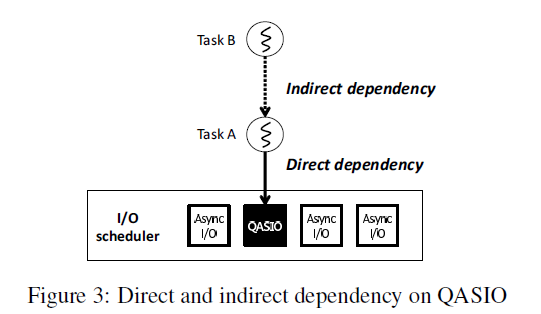
每个任务都可以与QASIO有直接或间接的依赖关系。 当任务执行由于（准）异步I / O被阻塞时，直接依赖关系发生。 图3说明了任务A直接依赖于QASIO的情况。 为了确定何时存在这种依赖关系，我们对Linux内核和文件系统调用产生的动态I / O模式进行了广泛的分析。 根据我们的分析，我们已经确定了以下四种QASIO的直接依赖关系类型：

修改元数据页面()时：当任务调用修改元数据页面（例如inode，组描述符，块位图，inode位图和Ext4中的目录项）的文件系统调用时，可能会发生此类型的依赖关系。被自己或其他任务标记弄脏的目标元数据页面，可能已经被kworker线程作为异步I / O提交。

修改数据页面()时：当任务在数据页内部分追加数据时可能会被阻塞，因为目标数据页面可能已被kworker线程异步刷新。数据页面到达存储区之前，任务不能继续执行。

确保数据写回()时：通过fsync()或truncate()同步或截断先前发送的文件数据时，任务需要等待异步I / O的完成。执行fsync()时,之前的所有缓存写入会同步发出，只要它们仍在页面缓存中。如果调用fsync()延迟或页面缓存中的脏页太多，则其中一些可能已经作为异步I / O被刷新。在这种情况下，fsync()需要等到这些异步发布的I / O完成。

当执行丢弃命令()时：目前，jbd2内核线程对于释放的块异步地发出丢弃命令，这与其他同步发布的日志块不同。 因此，每个日志提交都会阻止其执行，直到完成所有丢弃命令。 这个延迟反过来会影响前台任务的响应（参见）。



有时，一个任务的执行也可能因为另一个与QASIO有直接依赖关系的任务而被延迟。 例如，图3显示因为任务A直接依赖于QASIO而不能取得任何进展，任务B被阻塞。 在这种情况下，我们称该任务B对QASIO有间接的依赖关系。 通常情况下，当持有任务B所需资源的任务A被阻塞时就会出现这种情况。 与直接依赖不同，我们很难列出所有可能的间接依赖类型，因为由QASIO引起的延迟可以以不同的复杂的方式传播到其他任务。 然而，我们发现了以下两种与JBD2日志相关的间接依赖关系，它们对性能有很大影响。

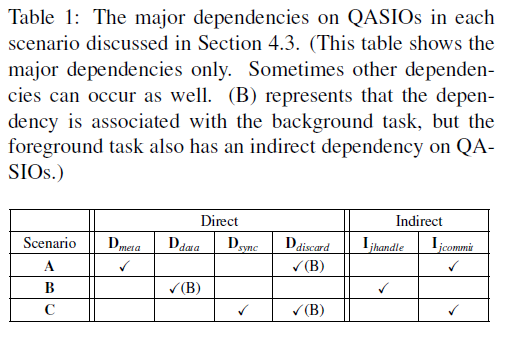
当由于或()而无法检索日志句柄时：在Ext4中，任务应该获取日志句柄以修改元数据页面或数据页面。如前所述，如果目标页面已异步发布则任务会被阻塞，这就产生了关于QASIO的或类型的依赖。当包含日志句柄的事务开始提交时，事务被锁定，因为被阻塞的任务持有日志句柄。在这种情况下，其他尝试执行任何文件操作的任务都会被阻塞，因为它无法获得新的日志句柄。

由于()而导致fsync()不能完成：只有在任务调用fsync()时，这种类型的间接依赖关系才会被察觉。 fsync()系统调用需要等待日志提交完成以确保相应文件的元数据已经被写入存储设备。但是，因为jbd2内核线程通常会由于异步发布的丢弃命令而产生类型的依赖关系，所以日志提交处理时间会有显着延长。

每当与用户交互的前台任务与QASIO有直接或间接的依赖关系时，执行过程中就会有非确定性的间断，用户会感觉到响应迟缓。 尽管在内存和请求队列中还有额外的I / O空间，系统调用的处理过程却被请求队列中的堆积的异步I / O阻塞。 更严重的问题是，由QASIO引起的滞后不仅发生在fsync()中，而且还会发生诸如creat()，chown()，unlink()甚至是buffered write()等用户并不希望发生延迟的系统调用中。 根据存储性能的不同，我们观察到由与QASIO的依赖关系导致的对creat()的单个调用需要的时间会增至几秒钟。

## 回顾问题场景

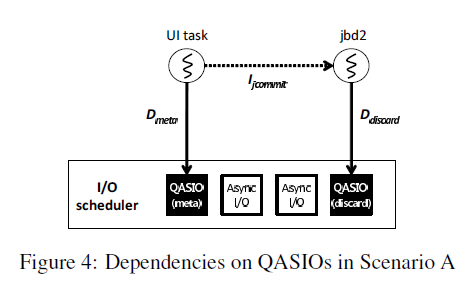
我们现在仔细看看第3节中有问题的场景以及它与QASIO的关系。表1总结了在每种场景中观察到的与QASIO的主要依赖关系。



### 场景A：启动CONTACTS应用

当一个应用程序的用户界面出现在Android平台上时，会进行多次文件系统调用将其状态不断地更新到数据库（例如SQLite）或文件（例如，xml文件）中。在我们的测试设备中，启动CONTACTS应用程序会伴随一系列系统调用，如rename()，write()，fsync()和unlink()。这些系统调用都需要更新元数据。因此，当它们尝试修改正在被写回的元数据页面时，在大量的异步I / O下，它们会与QASIO产生类型的依赖关系。

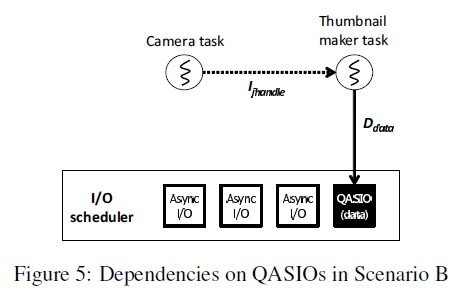
CONTACTS应用的UI任务具有与QASIO有关的另一种类型的间接依赖关系。在每个日志提交结束时，由于异步发布的丢弃命令，jbd2内核线程有直接依赖关系()。如果日志提交由于而延迟，那么由UI任务执行的fsync()系统调用也会延迟。因为在同步文件的元数据修改完全提交之前它无法返回。因此，UI任务与QASIO有两种依赖关系，如图4所示。



### 场景B：CAMERA应用的连拍模式

第一次连拍完成后，多项服务会通过从Android平台传输的Intents执行。 其中之一是生成拍摄图像缩略图的缩略图制作任务。 由于缩略图尺寸非常小，缩略图制作者任务重复小量地（例如，1KB）写入数据。 这会导致局部写入同一个数据页面，如果目标数据页面正在被写回，该数据页面可能被阻塞。 因此，缩略图制作任务与QASIO有类型的依赖关系。

由于缩略图制作者是在后台运行，因此不应该影响前台任务CAMERA的响应能力。 然而，问题在于CAMERA任务通过缩略图制作任务与QASIO有类型的间接依赖关系，如图5所示。缩略图制作任务在将缩略图数据复制到数据页面之前获取正在运行的事务的日志句柄， 并由于依赖关系陷入睡眠。 在一段时间之后，日志开始提交，但是jbd2线程陷入锁定状态，因为持有事务的日志句柄的缩略图制作任务由于进入了睡眠状态。 当CAMERA想要获取日志句柄以便为随后的连发镜头写入更多的图像时，它最终就会被阻塞。 这是因为在提交事务被锁定的情况下，JBD2日志模块没有给出任何日志句柄。



### 场景C：安装应用ANGRY BIRDS

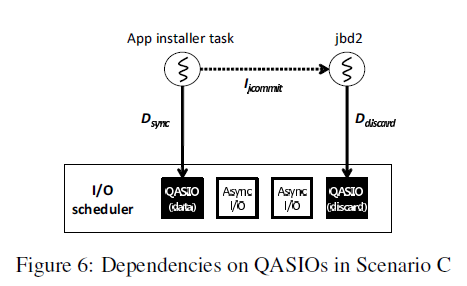


图6说明了在这种情况下出现的对QASIO的依赖关系。App安装程序任务会发出大量缓冲write()系统调用以便将下载的应用程序的提取数据保存到应用程序库。 为防止突然断电时的数据丢失，App安装程序任务调用fsync()将所有写入的数据刷新到存储设备中。 fsync()系统调用是用来同步写入属于相应文件的数据页面。但是，当大文件在后台写入时，页面缓存中将充满脏页面。 在这种情况下，应用安装程序调用fsync()之前，kworker线程可以异步刷新fsync()的数据页面。 这会导致与QASIO的类型依赖关系。

出于与场景A相同的原因，由于异步发布的丢弃命令，此场景也在fsync()期间有间接依赖关系。

# 提升QASIO

在本节中，我们将解释如何在运行时有效地检测QASIO，并在文件系统操作期间提高响应速度。

## 设计

当任务与QASIO有直接或间接依赖关系时，其执行将被阻塞直到相应的QASIO完成。 这种情况有点类似于调度中的优先级反转问题，高优先级任务因为低优先级任务具有其所需的资源而不能取得任何进展。在这种情况下，像优先级继承协议做的一样，我们可以采取的最佳方式是尽可能减少任务的等待时间，给予QASIO更高优先级并使其快速完成。

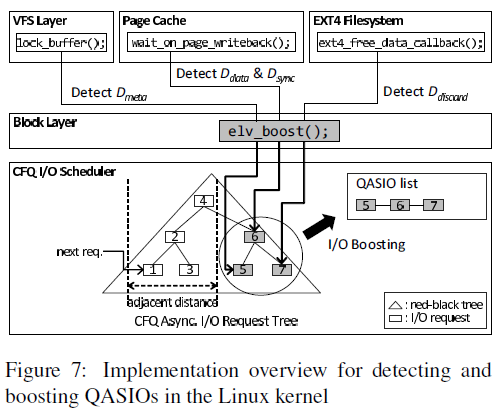
然而，在目前的I / O调度器的设计中这并不容易，因为他们不知道QASIO的存在，因此他们不知道应该提升什么。任务和QASIO之间的各种依赖关系是在运行时动态地在各种高层（如VFS，页面缓存和Ext4文件系统）上形成的。 这表明我们必须有一个运行时间机制，它可以检测到上层的QASIO并通知I / O调度器优先处理它们。

提升QASIO的要求可以总结为如下几条：

要求（1）：当某个任务被阻塞而等待异步I / O完成时，内核应该能够向I / O调度程序提示QASIO的存在。

要求（2）：在收到来自内核的提示后，I / O调度程序应该在异步I / O中优先考虑它们。

要求（1）独立于内核中使用的I / O调度程序，但是要求（2）需要为每种I / O调度器重新实现。 在本文中，我们只展示基于CFQ I / O调度器的实现。 但是，该设计可以很容易地应用于其他I / O调度程序。 图7概述了我们在Linux内核中实现QASIO的检测和提升的过程。



## 检测QASIO

在本小节中，我们介绍如何在运行时检测到Linux内核中的QASIO。 由于对QASIO的间接依赖只会在对同一个QASIO有直接依赖的情况下发生，因此我们只关注检测对QASIO的直接依赖关系。 如果解决了直接依赖关系，那么关联的间接依赖关系也将终止。

正如我们在4.2节看到的那样，关于QASIO有四种类型的直接依赖关系。 检测每种依赖关系的方式如下：

检测：在Ext4中，任务通过关联的缓冲区头结构访问元数据页面。 在任务修改元数据页面之前，它使用lock\_buffer()内核函数获得对元数据页面的排他锁定。 但是，如果缓冲区头已经提交到块层，则缓冲区头的状态变为锁定状态，并且尝试获取相同锁的任务的执行被暂停。 请注意，获取锁定失败并不一定意味着目标元数据页面作为异步I / O被提交。 因此，在做出决定之前，我们必须再次确认锁定的缓冲区头是否是作为异步I / O被处理。

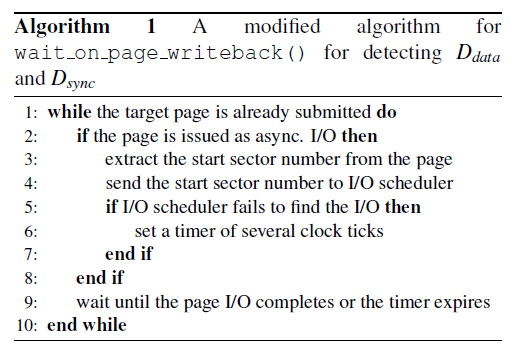
检测和：在Ext4中可以在同一位置检测到和的依赖类型。 当一个任务因为同步或截断一个文件要保证一些数据被写回或者执行一个数据页的部分写操作时，它会使用内核函数wait\_on\_page\_back()检查先前发出的I / O是否已经到达存储设备。 如有必要，该任务将在此函数过程中等待直到前一个I / O完成。 类似于的情况，我们应该检查之前的I / O是否作为异步I / O提交。

检测：内核函数ext4\_ free\_ data\_ callback()作为回调函数被注册到JBD2日志模块。 在日志提交结束时需要块取消分配时，此函数被调用。与上述两种情况不同的是，jbd2内核线程直接在回调函数中提交异步丢弃操作，然后进入睡眠状态等待这些I / O的完成。 显然jbd2生成了QASIO，所以不需要额外的检查。

要检测lock\_buffer()和wait\_on\_page\_writeback()函数中的QASIO，我们应该能够快速确认正在访问的缓冲区头或页面是否作为异步I / O发出。 由于Linux内核无法表示这些信息，因此我们添加了一个特殊的缓冲区标志并修改了submit\_bh()和ext4\_bio\_write\_page()函数，以便在提交异步I / O请求时设置该标志。 submit\_bh()和ext4\_bio\_write\_page()分别用来提交一个缓冲区头和一个数据页到下面的块层。这个特殊的标志在I / O完成后不会被复原。

算法1概述了如何在函数中实现对QASIO的实际检测。在原来的实现中，一个任务在wait\_on\_page\_writeback()函数过程中等待，直到目标页面的写回完成。在算法1中，首先检查目标页是否作为异步I / O提交（第1-2行），如果是这种情况，则将检测到的QASIO的扇区号发送到I / O调度程序（第3-4行） 。请注意，即使I / O调度程序被通知存在QASIO，也可能无法在请求队列（第5行）中找到它。原因如下：首先，I / O请求可能已经被低级驱动设备分派到存储设备。在这种情况下，除了等待I / O完成之外别无选择。第二种情况是

I / O请求暂时在任务的堵塞列表中，而不是在I / O调度器的请求队列中。堵塞列表保留了任务在任务堆栈空间上短时间产生的I / O请求，以增加创建更大请求的可能性，减少I / O调度器中的锁争用[3]。 由于堵塞列表是一个不能被搜索的私有区域，于是我们定期进行检查，直到插件列表中的所有I / O请求被刷新到请求队列。在lock\_buffer()和ext4\_ free\_ data\_ callback()函数中可以类似地检测到QASIO。



## 提升QASIO优先级

由于QASIO需要紧急处理，因此我们将QASIO的优先级调整到高于所有（真正的）异步I / O但不超过任何其他同步I / O。 这是因为我们不希望在提升QASIO时干扰同步I / O的响应。 处理QASIO的实际实现情况如下。

一旦检测到QASIO，QASIO的信息（即起始扇区号）应该被传送给I / O调度器。 为此，我们在Linux内核的电梯层中添加了一个名为elv\_boost()的新接口（参见图7）。elv\_boost()是一个抽象接口。它调用每个I / O调度程序特定的预注册函数。在我们的CFQ I / O调度程序中此函数为cfq\_boost\_req()。

对于每个异步队列，我们都维护一个名为QASIO列表的单独的I / O请求列表，如图7所示。在cfq\_boost\_req()中，我们遍历所有异步队列的红黑树并查找包含接收到的QASIO的扇区号的I / O请求。如果找到，则将QASIO的条目插入到相应的QASIO列表中。

在原始CFQ调度程序中，每当选择一个异步CFQ队列来调度请求时，CFQ会从红黑树中最后处理的请求中找到最近的请求（由next\_req指针指定），然后从此依次调度I / O请求。但是，在我们的实现中，CFQ首先检查QASIO列表然后分派QASIO请求（如果有的话）。此外，如果QASIO列表为空，则当前异步队列将产生将I / O分派到另一个具有非空QASIO列表的异步队列的机会。这样，QASIO就会在任何其他异步I / O请求之前进行分派。

# 评估

本部分通过五个微基准问题测试，三种真实场景以及两个Android的I / O基准测试来展示评估结果。

## 方法论

我们在最新的智能手机之一三星Galaxy S5上进行了评测，这款手机搭载了Exynos 5422（包括四核Cortex-A15和四核Cortex-A7 ARM处理器以及Mali-T628 MP6 GPU），2GB DRAM和16GB eMMC闪存。 它运行Android 4.4.2版本平台（KitKat），Linux内核3.10.9。 在Android系统中，脏页面过期时间设置为2秒，默认情况下后台脏比例为5％。

为了调查QASIO对各种文件系统操作延迟的影响，我们使用了以下内部微基准测试：

M1：M1重复500次创建4KB文件。 在每次迭代中，M1使用creat()打开相同的文件，然后使用缓冲write()将4KB数据写入文件。 最后，它执行fsync()并用close()关闭文件。 请注意，这个微基准测试模仿了数据库系统的存储I / O模式，如SQLite [12]。

M2：M2与M1相同，只是文件大小增加到1MB，迭代次数设置为200。文件数据（1MB大小）通过单个write()系统调用写入。

M3：M3用creat()创建一个新文件，并重复1KB大小的write()直到文件大小达到300MB。

M4：在每次迭代中，M4使用truncate()将上一次迭代中创建的2MB文件截断为零，随后使用write()重新创建相同大小的文件，然后使用close()关闭文件。 这个过程重复500次。 第一次迭代的文件是在执行之前手动创建的。

M5：M5通过执行creat()，write()，fsync()和close()来创建单个4KB文件，而另一个任务则截断现有的8GB文件并将8GB数据再次写入文件。

我们运行除M5之外的所有微基准测试，同时在后台写入8GB文件以生成异步I / O操作。 除了这些微基准测试之外，所提出的方案还将在第3部分中讨论的真实场景以及Android提供的两个具有代表性的I / O基准测试Antutu和RLBench中进行评估。

## 微基准测试

图8根据增强依赖的类型比较了每个微基准测试的总耗用时间。 结果标准化为NONE，其中QASIO没有执行特殊处理。 每个 + ，和表示检测并提升的特定类型的依赖关系的情况。 请注意，和类型不能单独提升，因为它们在同一位置被检测到（参见第5.2节）。 最后，ALL意味着所有类型的优化都被应用于QASIO。表2显示了在QASIO优化之前和之后的关键文件系统操作的延迟。总的来说，我们可以看到提升QASIO使得总耗时改进了83.1％。 所提出的方案还将每个文件系统操作的最坏情况等待时间减少了98.4％。 每个微基准测试结果的详细分析如下。

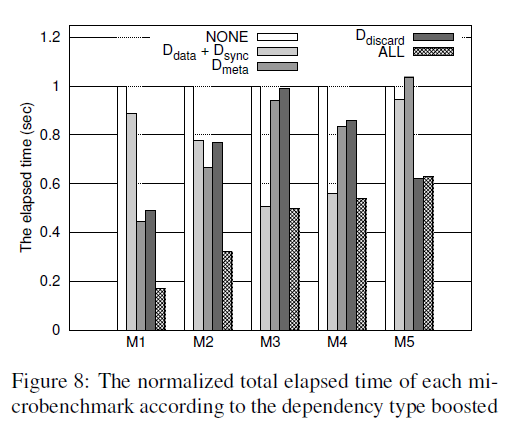
在M1中，当所有依赖类型被提升时，由于creat()和fsync()的平均延迟分别改进了99.1％和63.7％，总的运行时间减少了83.1％。 在M1的每次迭代中，creat()修改元数据页面并在创建具有相同名称的文件时引发丢弃操作。 因此，creat()和fsync()分别具有和依赖关系。 这就是为什么当和类型被提升时，总时间减少最多的原因。

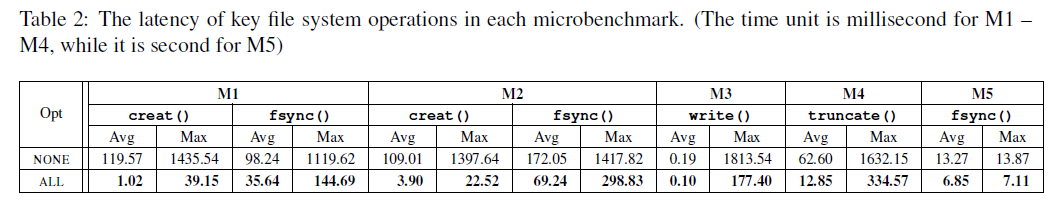
M2的结果显示了与M1类似的趋势，即提升 + 与提升或一样有效。 随着文件的大小增加，在调用fsync()之前，kworker可能会异步刷新某些数据页面。 因此，与M1不同，fsync()将在M2中具有依赖关系。

在M3中，影响整体性能的最主要的操作是缓存write()。从表2可以看出，write()的延迟平均下降了47.4％，最坏的情况下下降了90.2％。因为1KB的数据局部写入数据页面，write()会受到依赖关系的影响。因此，如图8所示，大部分性能改进是通过提升类型的依赖关系来实现的。

在M4中，受QASIO影响的关键文件系统操作是truncate()。如果文件数据已经异步发布，truncate()就要等待这些异步I / O的完成。因此，提升类型的依赖关系对于M4是最有帮助的。

在M5的情况下，jbd2对异步发出的丢弃命令有类型的依赖关系，因为另一个任务会截断大文件。在这种情况下，依赖关系导致调用fsync()来同步仅4KB的数据平均需要13.27秒。但是，当我们提升类型时，延迟会降低到6.85秒。



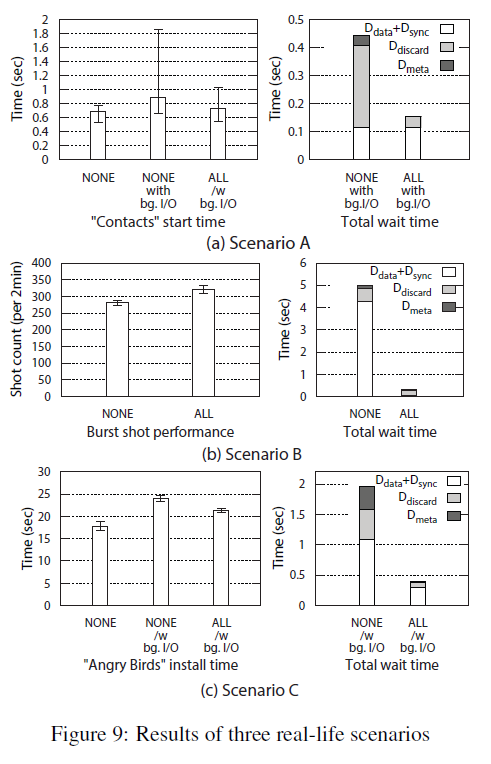


由于QASIO优先于I / O调度程序中的其他异步I / O，因此提升QASIO可能会对异步I / O的吞吐量产生负面影响。为了研究这个影响情况，我们测量了在执行M1微基准测试时在后台创建8GB文件的吞吐量。 根据我们的测量结果，吞吐量从46.2MB / s降低到39.1MB / s，降低了15.4％。 我们认为这是可以接受的，因为M1中前台任务的总时间改进了83.1％。

## 现实场景

图9描述了在第3节中介绍的现实场景中提升QASIO的影响。在图9的右侧，我们还显示了等待QASIO完成所花费的总时间。这些时间是在第5.2节描述的内核函数中测量的，其中每种类型的依赖关系都进行了检测。在图9（a）和（c）中，短语“with bg. I/O”表示同时在后台创建4GB文件以生成异步I / O的情况。

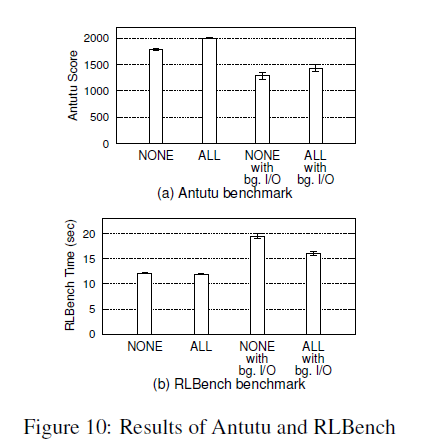
在情景A中，我们测量了CONTACTS启动应用程序的时间。在繁重的异步I / O下，启动时间平均增长了29.4％。在最糟糕的情况下，应用程序的启动速度慢了2.4倍。不过，QASIO有效地将最坏情况下的启动时间减少了44.8％。与M1微基准测试相似，最大的改进来自提升和，因为CONTACTS任务对QASIO有和类型的依赖。 和相应内核函数的总等待时间分别减少了96.1％和87.4％。



在情景B中，我们可以看到，当我们增加QASIO时，连拍模式下的镜头数平均提高了14.4％。缩略图制作任务对QASIO有类型的依赖，但由于造成的总等待时间减少了98.4％。因为缩略图制作任务与QASIO之间的直接依赖关系迅速得到解决，CAMERA应用程序的连拍模式性能也得到了改善。

最后，ANGRY BIRDS应用程序的平均安装时间在后台繁重的异步I / O下减慢了35.0％。 然而，我们观察到，通过提升QASIO平均安装时间改进了11.5％。 如第4.3.3节所述，ANGRY BIRDS应用程序与QASIO有和类型的依赖关系。因此，安装时间的减少大部分来自于提升。 提升和也有助于缩短安装时间，因为在应用程序安装过程中会执行一些元数据修改操作和丢弃操作。

## Android的I / O基准测试



最后，我们用两个著名的AndroidI / O基准测试，Antutu [6]和RLBench [15]，评估了提出的方案。 Antutu是一个全面的Android基准测试程序，提供了多种测量CPU，内存，存储系统等性能的模式。在此评估中，我们使用数据库IO模式来估计数据库工作负载下的存储I / O性能。 Antutu基准测试给出性能评估的最终分数，分数越高意味着性能越好。

RLBench是衡量SQLite工作负载下Android设备性能的基准测试程序。 它使SQLite生成存储I / O密集型工作负载，因此RLBench的结果与存储I / O性能密切相关。 与Antutu不同的是，RLBench显示处理预定义的SQL查询所用的时间，因此时间越短意味着性能越好。

Antutu和RLBench的结果显示在图10中。我们在有（标记为“with bg. I/O”）和没有异步I / O操作的情况下分别运行每个基准测试程序。运行基准测试程序时，通过在后台写入4GB文件生成异步I / O。

在Antutu中，当在后台没有异步I / O操作时，基础分数测量值为1785。异步I / O使得分降至1289。但是，该方案将分数提高到了1421分，比基准分数小了20.4％。一个有趣的现象是，即使在后台没有异步I / O的情况下，提升QASIO也会将Antutu的测试结果改进12.0％左右。这意味着Antutu本身发出异步I / O，其性能也受到QASIO的影响。

RLBench的结果也表明，该方案成功地提高了存储I / O性能。当在后台有异步I / O操作时，通过提升QASIO运行时间减少17.1％。

# 相关工作

移动设备通常采用NAND闪存作为主存储系统的媒介。 Kim等人通过对各种闪存存储系统的大量实验得出存储性能是移动设备常见应用性能的一个限制因素[9]。 由于NAND闪存与硬盘驱动器相比显示出非常不同的特性，所以之前的工作试图回顾已经针对旋转介质进行了优化的各种操作系统机制和策略。作为努力的结果，研究人员提出了针对闪存的特性而优化的多个文件系统[4,13]和I / O调度器[14,17]。

最近，许多研究集中在根据SQLite的I / O特性优化Linux内核的I / O栈。SQLite是由Android平台提供的轻量级事务数据库引擎。 由于大多数应用程序大多数利用SQLite来持久保存其特定于应用程序的数据，因此已知移动应用程序的整体性能高度依赖于SQLite的性能。 特别是，Lee等人观察到在Ext4文件系统之上运行SQLite会导致存储系统的I / O模式非常低效[12]。 基于这个观察现象，Jeong等人提出消除不必要的元数据日志，外部日志和基于轮询的I / O机制来提高日志效率[8]。 同样，Shen等人在SQLite环境中提出Ext4文件系统增强日志机制来解决所谓的日志的日志问题[18]。

在本文中，我们关注的是当系统有大量的异步存储I / O操作时，文件系统操作的响应性会严重降低。这是处理I / O请求时的固有问题，独立于文件系统和I / O调度程序。因此，我们的方法在很大程度上与以前的研究正交。需要注意的是我们提出的方案成功地减少了fsync()的延迟，而这个函数被SQLite频繁调用。因此，提升QASIO也将有助于提高SQLite的性能，以RLBench为例。

最近的Linux内核允许通过禁用稳定页面功能来更新在回写的数据页面，这成功消除了所有的依赖[5]。但是，由于在I / O期间的硬件支持加密正在被认真考虑，所以在将来的移动设备中这可能是不可接受的。另外，如最近Android平台[1]中介绍的，当用户空间程序在设备空闲的方便时间以批处理方式发出fstrim命令时，类型的依赖关系就被消除。但是，这不是一个完整的解决方案。因为如果在适当的时候没有发出丢弃命令，底层闪存的I / O性能可能突然下降。

# 结论

本文介绍了一种称为准异步I / O（QASIO）的新的I / O类型。 QASIO是看起来是异步但具有同步属性的I / O操作。因为有一个或多个任务被阻塞直到这种I / O操作完成。由于系统以最大化吞吐量而不是延迟的角度处理异步I / O的，被阻塞的任务的响应速度明显降低。特别是在大多数应用程序与用户始终交互的移动设备中，用户体验质量受到QASIO的影响。

为了解决这个问题，我们提出了一个新的方案来检测QASIO并在Linux内核中提升它们。我们已经在最新的基于Android的智能手机Samsung Galaxy S5s上实施并评估了的提出的方案。通过执行各种微基准测试，现实场景测试和Android I / O基准测试i，我们确认提升QASIO对提高文件系统操作的响应性是有效的。我们计划分析提升QASIO在包括服务器和低端智能手机在内的更多不同平台的影响。

视频编码标准发展到现在，一直不断地更新与改善，之前人民都以为H.264/AVC会是视频编码标准的最终一代，而近年来又发布了HEVC，国内的AVS二代等标准。HEVC被报告其压缩效率比上一代的H.264提高了百分之五十以上，这让科研人员产生了极大的兴趣，同时接着保持对编码技术潜心探讨的心，探寻性能提高的源头，仍然有关键的作用。

我们先是对优化技术做了简要的介绍，在编码标准制定后，它的大部分技术不可以再做变动，但是码控算法技术作为视频标准中的一项编码优化工具，不受限制，而且对提高压缩效率有着很大的作用，特对于，HEVC等标准由于是这几年才制定的，码控算法的呀就还比较少。虽然以前的H.264等视频标准对于这一块的研究在如今有一定的研究价值，然而HEVC公布了许多新的编码技术，这对科研人员带来了新的挑战。

接着，我们提出了一种基于二次编码的率失真优化方法，目的是保持视频质量平滑行。本文首先通过神经网络程序对视频进行第一遍编码，并得到预测的率失真模型，根据第一遍编码预测率失真模型，进行第二遍编码，有了第一遍的信息，再进行分配每一帧的比特数，在码率的限制下，到达更加平滑的视觉质量效果。

参考文献

[1] Android 4.3 update brings trim to all nexus devices.

http://www.anandtech.com/show/7185/android-43-update-brings-trim-to-all-nexus-devices.

[2] Android developers. http://developer.android.com.

[3] Explicit block device plugging. http://lwn.net/Articles/438256.

[4] F2fs: Introduce flash-friendly file system.

https://lwn.net/Articles/518718/.

[5] Optimizing stable pages. http://lwn.net/Articles/528031.

[6] ANTUTU LABS. AnTuTu Benchmark.

https://play.google.com/store/apps/details?id=com.antutu.ABenchMark5.

[7] GARTNER, INC. Gartner says worldwide traditional PC, tablet, ultramobile and mobile phone shipments are on pace to grow 6.9 percent in 2014. http://www.gartner.com/newsroom/ id/2692318.

[8] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In USENIX ATC (2013), pp. 309–320.

[9] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In FAST (2012), pp. 1–14.

[10] KIM, H., AND SHIN, D. Optimizing storage performance of android smartphone. In ICUIMC (2013).

[11] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In FAST, pp. 273–285.

[12] LEE, K., AND WON, Y. Smart layers and dumb result: IO characterization of an android-based smartphone. In EMSOFT (2012), pp. 23–32.

[13] LU, Y., SHU, J., AND WANG, W. ReconFS: a reconstructable file system on flash storage. In FAST (2014), pp. 75–88.

[14] PARK, S., AND SHEN, K. FIOS: a fair, efficient flash I/O scheduler. In FAST (2012), pp. 1–15.

[15] REDLICENSE LABS. RL Benchmark: SQLite.

https://market.android.com/details?id=com.redlicense.benchmark.sqlite.

[16] SAMSUNG. How do i use multi window mode (multitasking) on my Samsung Galaxy Note II?

http://www.samsung.com/us/support/howtoguide/N0000004/8829/62396.

[17] SHEN, K., AND PARK, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In USENIX ATC (2013), pp. 67–78.

[18] SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free. In FAST (2014), pp. 287–293.

[19] WIKIPEDIA. Burst mode (photography).

http://en.wikipedia.org/wiki/Burst\_mode\_ (photography).