对于ppt的引入主要从以下方面，先通过叙述现行环境引入超级文件系统EROFS,在通过叙述压缩文件系统和之前的文件系统，介绍他们的不足。再通过长篇的论证介绍EROFS的特点。再通过实验论证EROFS系统各方面的表现特点。

**问：EROFS是什么？**

EROFS：资源匮乏智能手机通常具有有限的存储空间和运行时内存。压缩的只读文件系统可以大大减少只读系统资源使用的存储量。但是，现有的压缩只读文件系统使用固定大小的输入压缩，这会导致显着的I / O放大和不必要的计算。它们还会在解压缩期间消耗过多的运行时内存，并在运行时内存不足时降低性能。

**问：只读文件系统与读写文件系统的不同**

与压缩的读写文件系统复杂且数据修改不同，压缩的只读文件系统通过设计排除了数据更新，这为更高的压缩率和更快的数据读取提供了更多机会

**问：EROFS设计的目的**

为了节省存储空间并以较低的内存开销保持高性能,我们设计并实现了EROFS，EROFS是具有压缩支持的增强型只读文件系统。 EROFS引入了固定大小的输出压缩，该压缩将文件数据压缩到多个固定大小的块，以显着减轻读取放大问题并尽可能减少不必要的计算。

**问：内存消耗的原因**

内存消耗。缺陷是Squashfs在解压缩期间需要大量的临时内存 压缩算法对所有原始数据进行解压缩，并将它们放入临时输出缓冲区中。最后，Squashfs将原始数据从临时输出缓冲区复制到相应的页面缓存页面。根据上述例程，使用了两个预分配的临时缓冲区

**问：输入压缩和输出压缩不同**

与固定大小的输入压缩相比，使用固定大小的输出压缩有很多好处。首先，正在相同压缩单位大小下，它具有更好的压缩率。因为固定大小的输出压缩可以最大程度地压缩数据，直到达到输出限制为止，而固定大小的输入压缩一次只能压缩固定大小的数据。其次，在解压缩期间，仅包含请求数据的压缩块将被读取和处理。

**问：Cpu缓冲区的缺点**

每CPU缓冲区解压缩上面的解压缩方法导致两个问题。第一个是仍然需要动态分配物理内存页面，这增加了内存受限设备的内存压力。第二个问题是，在每次解压缩时使用vmap和vunmap效率不高。

**问：为什么会要优化**

索引内存优化EROFS可能会将数百页的原始数据压缩到单个压缩块中。在这种情况下，EROFS需要数百个指针来跟踪应存储原始数据的每一页的位置。这些指针会占用大量内存。

**问：Brtfs和EROFS的工作特点**

例如Btrfs [2]，当压缩数据已被提取到内存中时，将唤醒一个专用线程来解压缩数据。完成解压缩后，将唤醒发出I / O的读取器线程，以从页面缓存中获取解压缩的数据。为了减少调度开销，EROFS在没有专用线程进行解压缩的情况下，在读取器线程中解压缩数据。

**问：图7显示出那些情况**

图7显示了我们观察到的以下结果。Btrfs在所有测试中表现最差，因为它既不是为压缩也不是为只读数据而设计的。一方面，Btrfs无法利用只读属性，而必须考虑更新另一方面，与在读取过程中不需要解压缩数据的Ext4和F2FS相比，Btrfs中的解压缩会带来显着的性能开销图9（a）和图9（b）显示了每个文件系统上用于enwik9和silesia.tar的字节数。该图还符合以下事实：压缩单位越大，压缩率越好

**问：个人总体感受与总结**

引入EROFS，这是一种新的压缩只读文件系统，专为资源有限的智能手机而设计。与Squashfs相比，EROFS提供了可比的压缩率，同时具有更高的性能和更少的额外内存开销。

与不带压缩支持的文件系统相比，通过固定大小的输出压缩以及快速且内存有效的解压缩，EROFS可以以较少的存储使用量存储系统代码和资源，有时甚至具有更好的性能。

与Ext4相比，EROFS上安装的系统上的应用程序可以启动甚至更快。 EROFS已合并到主线Linux中，并已在数千万智能手机中部署和使用。

个人认为系统应该增加例如重复数据删除，扩展文件统计信息，文件映射等功能(文章未提到)。

**附论文主要内容翻译：个人觉得第二章叙述超级文件系统和简介文件系统以及实验论证是本文主要核心部分**

# 2.EROFS:增强的压缩文件系统

本节介绍EROFS的设计，这是一种可压缩的只读文件系统，可克服现有系统的不足。 EROFS的关键设计包括固定大小的输出压缩，缓存的I / O和就地I / O以及内存有效的解压缩。

## 2.1固定大小的输出压缩

为了克服固定大小的输入压缩引起的读取放大，EROFS采用了不同的压缩方法：固定大小的输出压缩。为了生成固定大小的输出，EROFS使用滑动窗口压缩文件数据，该滑动窗口的大小是固定值，可以在生成图像时进行配置。多次调用压缩算法，直到所有文件数据都被压缩为止。例如，对于1MB的滑动窗口，EROFS一次向压缩算法提供1MB的原始数据。然后，该算法将尽可能压缩原始数据，直到使用完所有1MB数据或使用的数据可以生成正好4KB的压缩数据为止。剩余的原始数据将与更多数据合并，形成另一个1MB的原始数据，以供下次调用压缩时使用。图2（c）描述了固定大小的输出压缩，其中可变大小的原始数据被压缩为4KB块。与固定大小的输入压缩相比，使用固定大小的输出压缩有很多好处。首先，正如我们在评估中将看到的（第5.3节）所示，在相同压缩单位大小下，它具有更好的压缩率。这是合理的，因为固定大小的输出压缩可以最大程度地压缩数据，直到达到输出限制为止，而固定大小的输入压缩一次只能压缩固定大小的数据。其次，在解压缩期间，仅包含请求数据的压缩块将被读取和处理。在前面的示例中，请求单个原始块时，最多将读取和解压缩两个压缩块。第三，正如我们稍后将在§3.3中显示的那样，固定大小的输出压缩使就地解压缩成为可能，这进一步降低了EROFS中的内存消耗。

## 2.2缓存的I / O和就地I / O

在实际解压缩之前，EROFS需要空间来存储从存储中检索到的压缩数据。尽管由于过多的内存分配甚至页面交换，对于固定大小的输入压缩来说这是昂贵的，但是固定大小的输出压缩所产生的成本要少得多，因为EROFS清楚地知道每个压缩最多只能检索两个压缩块。 EROFS有两种策略：缓存I / O和就地I / O。 EROFS将缓存的I / O用于将被部分解压缩的压缩块。 EROFS管理一个特殊的索引节点，其页面高速缓存存储按物理块编号索引的压缩块。因此，对于高速缓存的I / O，EROFS将在特殊索引节点的页面高速缓存中分配一个页面以发起I / O请求，以便存储驱动程序将压缩数据直接提取到分配的页面中完全解压缩后，EROFS使用就地I / O。在每次读取文件时，VFS都会在页面高速缓存中为文件系统分配页面以放置文件数据。对于这些页面中的任何一个，在减压之前均不包含任何有意义的数据，我们将其称为可重用页面。对于就地I / O，EROFS使用最后一个可重复使用的页面来初始化I / O请求。两种I / O策略都是必需的。对于缓存的I / O，部分解压缩的块将缓存在特殊页面缓存中，以便后续对未压缩部分的读取可以使用这些块，而无需调用其他I / O请求。对于完全解压缩的块，它们不太可能在以后使用，因为所有解压缩的数据都存储在页面缓存中，该页面缓存可用于后续读取而无需解压缩。因此，由于对完全压缩的块进行页面分配，所以缓存的I / O毫无用处地增加了内存峰值，同时又不影响后续的文件读取。在这种情况下，就地I / O避免了不必要的内存分配，从而减轻了内存压力，尤其是当在不同压缩块上有许多空文件读取请求时。请注意，尽管可以将压缩块放在堆栈上，但不建议这样做，因为堆栈大小限制为16KB [14]，并且不容易知道堆栈中还有多少字节可用。

## 2.3减压

将压缩的数据加载到内存中之后，我们将说明EROFS如何快速而有效地对数据进行解压缩。 本节中的示例基于图3（a），其中前五个块（D0至D4）和部分D5压缩为块C0，其余块压缩为块C1。 在本小节中，我们仅介绍

单个压缩块如何解压缩，因为对于包含多个压缩块中数据的读取请求，压缩块类似地逐个解压缩。例如，要读取图3（a）中的块D4至D6，首先将C0解压缩以获得D4和D5的第一部分；然后将C1解压缩以获得D5和D6的其余部分。  
Vmap解压缩要获取块D3和D4中的数据，EROFS首先从存储器中读取压缩块C0并将其存储在存储器中。然后EROFS将按照以下步骤对其进行解压缩。  
1.找到存储在压缩块（C0）中的最大所需块号，在示例中为第五个块（D4）。作为优点，EROFS仅需要解压缩前五个块（D0至D4），而不需要解压缩所有原始数据块。

2.对于每个需要解压缩的数据块，找到存储空间以存储它们。在图3（b）所示的示例中，EROFS分配了三个临时物理页来存储D0，D1和D2。对于请求的两个块D3和D4，EROFS重用了由VFS在页缓存中分配的两个物理页。

3.由于解压缩算法需要连续的内存作为解压缩的目标，因此EROFS通过vmap接口将上一步中准备的物理页映射到连续的虚拟内存区域中。

4.如果是就地I / O，则在这种情况下压缩块（C0）存储在页面缓存页面中，EROFS还需要将压缩数据（C0）复制到临时的每CPU页面中，以便获得解压缩的数据在解压缩期间不会覆盖压缩的数据。

5.最后，调用解压缩算法，并将压缩块中的数据提取到连续存储区。压缩之后，可以收回三个时间物理页面和虚拟存储区域，并且所请求的数据已经被写入到相应的页面缓存页面中。

每CPU缓冲区解压缩上面的解压缩方法导致两个问题。第一个是仍然需要动态分配物理内存页面，这增加了内存受限设备的内存压力。第二个问题是，在每次解压缩时使用vmap和vunmap效率不高。当解压缩的数据少于四个页面时，EROFS利用每个CPU的缓冲区来缓解问题。如图3（c）所示，为每个CPU预先分配一个四页的存储器缓冲区作为每个CPU的缓冲区。对于解压缩最多提取四个数据块，EROFS将数据解压缩到每个CPU缓冲区，然后将请求的数据复制到页面缓存页面。在图3（c）所示的示例中，请求块D8中的数据。将C1中的压缩数据直接解压缩到每个CPU缓冲区，然后D8的内容通过memcpy复制到页面缓存页面。每个CPU缓冲区的长度由经验决定，但是由于每个CPU缓冲区可以在不同的解压缩中重用，因此它可以有效地消除内存分配。每CPU缓冲区解压缩是一种具有成本效益的折衷，可以减轻vmap解压缩中的问题，同时引入额外的内存副本。

滚动解压缩为避免vmap和vunmap的开销并消除其他动态页面分配，EROFS为每个CPU分配了一个较大的虚拟内存区域2和16个物理页面。每次压缩之前，EROFS都使用16个物理页面以及页面高速缓存的物理页面填充VM区域，因此可以跳过vmap解压缩的步骤2和步骤3。 EROFS使用LZ4作为压缩算法，该算法需要向后看不超过64KB的解压缩数据[7]。因此，对于提取多于16个页面的压缩，EROFS可以重用之前映射16个虚拟页面（即64KB）的物理页面。例如，在图4中，由16个物理页支持存储块D0至D15的虚拟地址。 D16的虚拟页面可以由与D0相同的物理页面支持，因为D16中的每个虚拟地址都与D0中的相应地址相距64KB。 D17由D1使用的物理页面以相同的方式支持。文件读取请求的D18使用页面缓存的物理页面。结果，通过使用这种滚动解压缩，对于任何解压缩情况，都有16个物理页就足够了

就地解压缩在vmap解压缩方法的步骤4中，已压缩的数据被移动到每个CPU的临时页面，以避免尚未压缩的数据被已压缩的数据覆盖（图5）。

图5：无法就地解压缩的示例压缩块（C0）。 序列A的解压缩数据破坏了尚未解压缩的下一个序列（阴影部分）。  
但是，如果压缩块永远不会发生这种情况，则可以就地对其进行解压缩，以避免额外的内存分配和内存副本。 EROFS模拟mkfs期间的解压缩，并标记是否可以在块索引中就位解压缩压缩的块。 在对可以就地解压缩的块进行解压缩期间，将跳过步骤4。 在§5中经过测试的工作负载enwik9中，可以就地解压缩99.6％的压缩块。 因此，只要通过就地I / O检索，大多数块都可以从就地解压缩中受益。

# 3.实施

我们已经将EROFS实施为Linux文件系统，并将EROFS的公共部分上游到Linux内核3。在当前的实现中，我们使用4KB作为固定的输出大小，因为它是页面管理和存储数据传输的最小单位，因此可以最大程度地减少I / O放大。我们支持LZ4（v1.8.3）作为压缩算法，因为它在我们的情况下具有最快的解压缩速度和良好的压缩率。一旦对其他压缩算法（例如LZO）进行了修改以提供固定大小的输出压缩接口，就可以支持它们。在EROFS中仅压缩文件数据。元数据（例如inode和目录条目）无需压缩即可存储。目前，EROFS仍在积极开发中，并且经过严格的商业测试过程后，新功能不断被带入智能手机。因此，我们介绍了EROFS的两个版本：1）具有本文介绍的所有功能和优化功能的最新版本； 2）商业部署的版本，具有除滚动解压缩和就地解压缩之外的所有功能和优化。这两个版本的减压策略也不同，我们将在第4.2节中说明。

## 3.1 EROFS图像布局

图6显示了EROFS图像的布局。与其他文件系统一样，超级块位于图像的开头。在超级块之后，元数据和数据可以以混合样式存储，而不受顺序的限制。在当前的实现中，文件的元数据和数据存储在一起，以实现更好的局部性。对于每个文件，如图6所示，索引节点存储在开头，然后是包含扩展属性（即xattrs）和块索引的块。压缩或未压缩文件数据的块（编码块）存储在每个文件的末尾。由于可以将inode放置在图像中的任何位置，因此可以根据inode的位置来计算inode的数量，以便可以快速定位inode。如果文件不包含xattrs或未压缩，则省略xattrs的块和块索引。此外，如果可能，还可以在索引节点中内嵌xattrs，块索引和文件数据，这可以减少存储开销并减少I / O请求的数量，因为内联的数据/元数据是与索引节点一起读取的。块索引用于为读取请求快速定位相应的编码块。图6显示了压缩前包含10个块的规则文件的示例块索引。块索引是8B长度条目的数组，每个条目对应于压缩前的数据块。每个条目指示相应的数据块是否是头块（图6中的布尔头字段），头块开始一个新的编码块。如果是，则编码块地址（blkaddr），新编码块中第一个字节的偏移量（偏移量），编码块是否已压缩（cmpr）以及该块是否可以就地解压缩（dip）。也存储。如果没有，则在未压缩的块之前必须有一个起始块，并且与起始块的块号差以dist记录。对于对未压缩数据块的读取请求，EROFS根据请求的块号获取块索引条目。对于头块，EROFS从blkaddr处的块解压缩数据，如果偏移量不为零，则EROFS可能还需要从blkaddr之前存储的最近编码块解压缩。对于非头块，EROFS根据存储的dist计算相应头块的位置，并开始解压缩，直到对请求的块数据进行解压缩为止。压缩后较大的一些数据块（例如，图6中的块5）未被压缩，而是直接存储为编码块。对于这些情况，将相应的cmfield字段设置为false（即，图中的“ N”）。

目录的存储方式与常规文件类似，只是没有块索引，并且已编码的块用于存储未压缩的目录条目。 为了更好地在目录中进行随机访问，EROFS将所有Dirent标头（例如，索引节点号，文件类型和名称长度）放在目录条目部分的开头，并将文件名放在这些标头之后。

## 3.2减压政策

EROFS的两个版本具有不同的解压缩策略。如果要提取的原始数据块少于四个，则商业部署的EROFS使用每CPU缓冲区解压缩功能；否则，将使用vmap解压缩。在最新的EROFS中，实现了所有四种减压方法。如果要提取的数据块不超过一个，则选择按CPU缓冲区解压缩。否则，如果使用就地I / O检索了压缩块并可以就地解压缩，则EROFS将采用就地解压缩方法，从而避免了不必要的内存分配和内存副本。在其他情况下，解压缩的块可以放在预分配的VM区域中，因此EROFS使用滚动解压缩，因为它击败了vmap解压缩，具有较少的内存分配开销。对于任何其他情况，都将使用vmap解压缩方法。

## 3.3优化

索引内存优化EROFS可能会将数百页的原始数据压缩到单个压缩块中。在这种情况下，EROFS需要数百个指针来跟踪应存储原始数据的每一页的位置。这些指针会占用大量内存。为了解决此问题，EROFS尝试在可重用页面的帮助下存储信息。如果有多个VFS分配的页面可重用，则EROFS在I / O期间使用最后一页存储压缩数据，并使用其他页面存储其中的某些指针。在实际解压缩之前，这些指针会移到堆栈上，以便可重用的页面可以自由存储解压缩的数据。计划优化减压需要相对较长的时间。因此，不适合在中断上下文中完成。在某些文件系统中，例如Btrfs [2]，当压缩数据已被提取到内存中时，将唤醒一个专用线程来解压缩数据。完成解压缩后，将唤醒发出I / O的读取器线程，以从页面缓存中获取解压缩的数据。为了减少调度开销，EROFS在没有专用线程进行解压缩的情况下，在读取器线程中解压缩数据。因此，一旦压缩数据已被提取到内存中，读取器线程将被直接唤醒并开始对数据进行解压缩。同类群组减压可以同时执行多个请求。如果在线程A上请求原始数据块，并且相应的压缩块正在由另一个名为线程B的线程解压缩，而不是自己对数据进行解压缩，则线程A可以等待线程B完成解压缩，然后直接读取从页面缓存中解压缩的数据。这样的协作重用了解压缩的数据，并防止了单个数据被多次解压缩。映像修补程序尽管EROFS是压缩的只读文件系统，但在某些情况下（例如系统升级或安全修补程序），需要更新EROFS中存储的数据。 EROFS提供了一种称为图像修补的功能，该功能支持部分数据更新。通常，修改原始文件数据中的单个位可能会导致压缩数据中发生大量分散的修改。图像修补不是在原位进行修改，而是将更新的数据放置在EROFS图像的末尾，并且当请求相应的文件数据块时，首先对原始数据块进行解压缩，然后应用更新的数据覆盖其中的解压缩数据。记忆。这样，图像修补可防止更改分散，并支持部分数据更新，而无需重新压缩文件系统。

# 4.评价

我们进行了一系列实验来回答以下问题：•压缩如何影响文件系统读取访问的性能？ •EROFS在解压缩期间会消耗多少内存？ •EROFS如何影响实际应用程序的启动时间？

## 4.1评估设置

默认情况下，我们在运行Linux 9内核和Android 9 Pie的ARM开发板HiKey 960上进行实验。该主板配备了Kirin 960（四个Cortex-A73大内核和四个Cortex-A53小内核），3GB Hynix LPDDR4内存和32GB Samsung UFS存储。在一些实验中，我们还评估了两种智能手机。低端智能手机配备了MT6765（八个Cortex-A53内核），2GB内存和32G BeMMC存储。高端智能手机运行Kirin 980（四个Cortex-A76内核和四个Cortex-A55内核），6GB内存和64GB UFS存储。对于微基准测试，我们在包括EROFS，Squashfs，Btrfs，Ext4和F2FS在内的各种文件系统上运行FIO [23]，这是一个灵活的I / O测试器。我们使用最新版本的EROFS进行微基准评估。在这些文件系统中，EROFS和Squashfs被设计为压缩的只读文件系统。 Btrfs是具有压缩支持的文件系统，但它不是为只读数据而设计的文件系统。 Ext4是Android使用的默认文件系统； F2FS是为移动设备设计的文件系统，在某些智能手机中得到了广泛的使用。 EROFS配置为对LZ4使用4KB大小的输出压缩。 Squashfs被配置为使用压缩块大小为4KB，8KB，16KB和128KB的LZ4，分别由Squashfs-4K，Squashfs-8K，Squashfs-16K和Squashfs128K表示。 Btrfs被配置为以只读模式运行，而无需进行数据完整性检查以进行公平的比较。由于Btrfs不支持LZ4，因此Btrfs使用的压缩算法是LZO。由于实验不支持Ext4和F2FS，因此它们都无需压缩即可使用。对于实际应用，我们将EROFS与Ext4进行了比较，因为Ext4现在是Android使用的默认文件系统[17]。我们将商业版本用于实际评估，因为将最新版本发布到智能手机需要时间。我们还尝试在Android上使用Squashfs。但是，它占用了过多的CPU和内存资源，并且在尝试运行摄像头应用程序时，手机冻结了数十秒，直到最终失败

## 4.2微观基准

我们使用FIO来展示不同文件系统的基本I / O效率。在本实验中，我们使用enwik9 [40]作为工作负载，这是英语Wikipedia转储的前109个字节。我们将文件存储在不同的文件系统中，并读取文件以测试文件系统的吞吐量。每个读取是4KB缓冲读取。我们在三种情况下测试吞吐量：顺序读取，随机读取和跨步读取。对于顺序读取，我们依次读取4KB x 4KB的文件。因此，由于先前的解压缩或预取（即预读）已经将数据加载到了内存中，因此以下读取很有可能会在缓存中命中。对于随机读取，我们随机读取整个文件。因此，如果先前的读取已经对数据进行了解压缩，则这些读取会进入缓存。最后一种情况是跨步读取，其中我们仅读取每128KB数据中的第一个4KB。由于最大的压缩块为128KB，因此跨步读取不会在cache4中命中。我们测试步幅读数以说明压缩文件系统的最坏情况性能。在每次测试之前，将删除页面缓存以减少干扰。所有测试至少完成十次，并报告平均吞吐量。对于A53磁芯上的步幅读数，最大相对标准偏差为17.3％，其余结果为5.1％。图7显示了我们观察到的以下结果。与EROFS和Squashfs-128K相比，Btrfs在所有测试中表现最差，因为它既不是为压缩也不是为只读数据而设计的。一方面，Btrfs无法利用只读属性，而必须考虑更新；因此，其性能优于压缩的只读文件系统EROFS和Squashfs-128K。另一方面，与在读取过程中不需要解压缩数据的Ext4和F2FS相比，Btrfs中的解压缩会带来显着的性能开销。这是合理的，因为Btrfs并非设计为压缩的只读文件系统。对于连续读取，Btrfs的性能优于Squashfs的其他配置，这是由其较大的压缩块（128KB）引起的。优点是在无法进行预取的随机读取中缩小；优势在跨步读取中消失了，在跨步读取中，解压缩比请求更多的数据成为负担。总的来说，此结果表明使用具有压缩支持的只读只读文件的通用文件系统效率低下，并强调了设计压缩只读文件系统的必要性。随着压缩输入大小的增加，对于随机读取和顺序读取，Squashfs的性能会增加，而对于跨步读取，Squashfs的性能会下降。造成这种现象的主要原因是位置和缓存。由于在此实验中文件系统具有足够的内存来缓存文件数据，因此所有解压缩的数据都将被缓存，并可能在将来读取。因此，对于随机读取和顺序读取，将对较大数据进行解压缩，将来会有更多读取进入缓存。这基本上是Squashfs吞吐量随压缩块大小增加而增长的原因。由于顺序读取和随机读取都将读取整个文件，因此性能差异很小，这是由良好的局部性和预取引起的。但是，对于跨步读取，FIO对于每个128KB数据仅读取第一个4KB数据，这消除了内存高速缓存的好处，因为将来解压缩但未请求的所有数据都将不再使用。因此，读取和解压缩的数据越不相关，浪费的时间和资源就越多，从而导致性能下降。这解释了为什么吞吐量随Squashfs压缩块大小的增加而下降。在具有压缩支持的文件系统中，EROFS在大多数测试中表现最佳，有时甚至胜过不压缩数据的文件系统。对于顺序读取，EROFS在压缩文件系统中表现出最佳性能。与Squashfs相比，多数优势来自固定大小的输出压缩设计以及消除不必要的内存分配和数据移动。对于随机读取，Squashfs128K优于EROFS，因为后者可以在测试期间解压缩并缓存整个文件，而EROFS仅从缓存的I / O中受益。但是，EROFS的性能仍然优于其他压缩文件系统。对于大步读取，由于预取几乎没有用，因此EROFS仍在压缩文件系统中产生最佳吞吐量，但胜利有限。与不支持压缩的Ext4和F2FS相比，EROFS的性能始终优于甚至优于它们（例如，在A73内核上的顺序读取）。原因是，即使EROFS需要解压缩数据，由于进行压缩，它也会从存储中读取少得多的数据。

我们还使用工作量silesia.tar [9]进行相同的实验。 Silesia.tar是西里西亚压缩语料库的压缩文件，其中涵盖了当今使用的典型数据类型。 结果显示出与enwik9相同的趋势，因此我们仅在图8中呈现A73内核在2362MHz处的结果。

## 4.3压缩率和内存使用率

我们还评估了每个文件系统解压缩期间的压缩率和内存消耗。 我们使用enwik9和silesia.tar来表示不同文件系统的压缩率。 图9（a）和图9（b）显示了每个文件系统上用于enwik9和silesia.tar的字节数。 两个图中的原始线代表未压缩的工作负载文件的大小，enwik9为953.67MB，silesia.tar为202.1MB。 当前，EROFS仅支持4KB大小的输出压缩，但是与Squashfs-4K相比，这两种工作负载的压缩大小分别小10％和9％。 该图还符合以下事实：压缩单位越大，压缩率越好

图10显示了解压缩enwik9文件后使用的内存。该测试按以下方式进行：引导计算机，安装文件系统，读取存储在文件系统中的文件，检查使用的内存，然后重新启动。由于文件大约为1GB，因此剩余部分将由操作系统的其他部分使用，或者由文件系统临时使用。除了EROFS和Squashfs，我们还测试了Ext4作为基准。从图可以看出，与Ext4相比，四个配置的Squashfs的内存开销为39.6％至61.6％。但是，EROFS使用的内存仅略高于Ext4（约4.9％）。结果表明，EROFS的内存峰值比Squashfs低得多，并证明了EROFS的内存友好型解压缩的有效性。在测试中，只有一个文件要解压缩，并且我们分配了足够的内存以确保在解压缩期间不会发生内存回收或交换。但是，在现实世界中，将同时解压缩更多文件，通过Squashfs的解压缩将需要更多的内存。一旦可用内存不足，可能会发生内存回收或交换，这非常广泛，不仅会影响文件系统，还会影响整个系统中的其他组件或应用程序。因此，在实际情况下，EROFS的优势会更加显着，它的优点是在解压缩过程中只需占用很少的内存。

## 4.4 I / O放大和I / O模式

我们重新运行了§2.2中关于EROFS和不同Squashfs配置的测试。表1列出了在三种读取模式下读取16MB文件数据时发出的实际I / O。 EROFS为随机读取和跨步读取发出的I / O最少。但是，由于Squashfs-8K，Squashfs-16K和Squashfs-128K具有更好的压缩率，因此连续读取的数据比EROFS少。总之，与Squashfs相比，EROFS在大多数情况下都减少了I / O放大，特别是对于随机读取和跨步读取。我们进一步在模拟的真实环境中确定了I / O模式，以说明I / O放大将如何影响实际应用。我们安装了100个应用程序，并运行了Monkey工具[13]，每秒随机点击一次屏幕3个小时。我们收集了传递给readpage和readpages接口的I / O大小，并在表2中显示了不同I / O大小的比例。结果表明，有很多I / O（30.4％），大小不超过16K，我们将其视为随机I / O。随机I / O的数量是合理的，因为随着系统长时间运行，由于内存不足，应用程序的页面缓存中的某些页面将被回收。少量的随机I / O强调了EROFS努力减少I / O放大的重要性

## 4.5吞吐量和空间节省

图11描述了在节省空间的情况下EROFS和Ext4的吞吐量。节省空间是通过压缩减少的空间值除以原始数据的大小；因此，节省的空间越大表示节省的空间越多。为简单起见，我们只显示大核心的结果，因为在小核心上也显示了类似的趋势。为了进行测试，我们在压缩分区中收集了块，并检查了它们的空间节省。当我们找到一个符合预期空间节省的块时，原始数据将被解压缩并重复多次，以形成大约512MB的文件。然后，我们将文件存储在EROFS中并对其进行读取，以在节省空间的情况下获得读取吞吐量。我们还将文件存储在Ext4中，并获得了相应的读取吞吐量以进行比较。通常，Ext4的吞吐量在测试过程中保持稳定，并且EROFS的性能提高，同时节省空间。当节省的空间足够高时，EROFS的吞吐量要比Ext4好得多。在这种情况下，单个压缩块可以解压缩为数十个块。因此，I / O请求的数量显着减少，从而导致更高的性能。当空间节省较少时，对于随机读取，EROFS的性能与Ext4相似，而对于顺序读取则比Ext4差。这是I / O成本和解压缩计算成本相结合的结果。对于随机读取，I / O比解压缩计算昂贵。对于顺序读取，由于预取，I / O的成本较低，并且当空间占用时，计算成本占主导地位

图11描述了在节省空间的情况下EROFS和Ext4的吞吐量。节省空间是通过压缩减少的空间值除以原始数据的大小；因此，节省的空间越大表示节省的空间越多。为简单起见，我们只显示大核心的结果，因为在小核心上也显示了类似的趋势。为了进行测试，我们在压缩分区中收集了块，并检查了它们的空间节省。当我们找到一个符合预期空间节省的块时，原始数据将被解压缩并重复多次，以形成大约512MB的文件。然后，我们将文件存储在EROFS中并对其进行读取，以在节省空间的情况下获得读取吞吐量。我们还将文件存储在Ext4中，并获得了相应的读取吞吐量以进行比较。通常，Ext4的吞吐量在测试过程中保持稳定，并且EROFS的性能提高，同时节省空间。当节省的空间足够高时，EROFS的吞吐量要比Ext4好得多。在这种情况下，单个压缩块可以解压缩为数十个块。因此，I / O请求的数量显着减少，从而导致更高的性能。当空间节省较少时，对于随机读取，EROFS的性能与Ext4相似，而对于顺序读取则比Ext4差。这是I / O成本和解压缩计算成本相结合的结果。对于随机读取，I / O比解压缩计算昂贵。对于顺序读取，由于预取，I / O的成本较低，并且在节省空间的情况下，计算成本占主导地位

## 4.6不同的减压方法和优化

为了说明不同减压方法的效果，我们在配备A76内核且频率为2600MHz的Kirin 980智能手机上运行了FIO顺序读取。 vmap解压缩方法的文件读取速度为726.5MB / s，而每CPU缓冲区解压缩的吞吐量为736.5MB / s。在最新的EROFS中，添加了滚动解压缩和就地解压缩后，文件数据的读取速度为769.7MB / s。我们还用相同的配置评估了第4.3节中调度优化的效果。在随机读取工作负载中，未经调度优化的EROFS的平均吞吐量为64.49MB / s，而通过优化，性能提高了9.5％，为70.61MB / s。

## 4.7实际应用

对于现实世界的应用程序，我们在低端智能手机和高端智能手机上都运行了经过修改的Android 9 Pie，它们的硬件配置在§5.1中列出。 Android系统分区（例如/ system，/ vendor和/ odm）已使用EROFS进行了压缩，节省的空间范围为30％至35％。我们测试了生产团队所需的13种流行应用程序的启动时间。我们将EROFS上的应用程序启动时间与Ext4上的应用程序启动时间进行了比较，并在表3中显示了相对启动时间。与Ext4相比，EROFS平均将低端智能手机的启动时间减少了5.0％，将高端智能手机的启动时间减少了2.3％。在运行FIO时，我们还进行了与后台工作负载相同的测试，以模拟实际场景。在FIO工作负载中，四个线程随机读取和写入单个文件，并且读取和写入的速率限制为256KB / s。表3的最后两行显示了带有FIO工作负载的启动时间，其中低端和高端智能手机的启动时间分别减少了3.2％和10.9％。  
除了各种应用程序的启动时间以外，我们还测试了上述高端智能手机上相机应用程序的启动时间分布。为了模拟内存不足的情况，我们在后台运行了一个程序，该程序连续分配内存并填充垃圾数据。在开始实验之前，我们一直等到程序耗尽了系统中的所有zram，并在评估期间保持其运行。在实验中，我们依次启动了多个应用程序，并记录了相机启动时的启动时间。我们为EROFS和Ext4每次收集了92个摄像机引导程序，并在图12中给出了累积分布。对于90％以上的引导程序，在EROFS上运行的摄像机应用程序比在Ext4上运行更快，而在OSFS上的启动时间最长。 EROFS比Ext4差。我们认为结果是可以接受的，因为在大多数情况下，EROFS可以节省存储空间，同时减少启动时间