移动设备文件压缩算法综述

胡可心[1)]

[1)]（华中科技大学计算机科学与技术系　武汉　430070）

摘　要　移动设备已成为人们日常生活中必不可少的一部分，人们在使用移动设备的过程中将会产生海量的数据，这导致了对移动设备存储和传输数据的高效技术的需求。例如，使用淘宝 APP 购买商品将会产生大量数据，包括浏览历史记录、购买记录、搜索记录等，这使得这些数据存储在有限的存储空间或通过 Internet 有限的带宽进行通信成为一个巨大的挑战。因此，对移动设备上的数据压缩和通信理论的研究越来越多，以应对这些挑战。通过对数据执行压缩操作可以永久或临时地减少数据的大小，在数字处理中，不仅要考虑数据规模，而且要节省时间。优化存储介质和带宽需要效益和压缩过程，满足特定的目标和需求需要不同压缩技术。移动应用程序表现出独特的文件访问模式，通常涉及对主要写入文件和只读文件的随机访问，移动设备程序严重依赖嵌入式数据库层 SQLite 进行事务数据管理，这将会产生大量的小文件覆盖和追加操作，高写入压力严重影响着基于闪存的移动存储的寿命。然而现有的数据压缩技术大多是面向传统桌面系统，这些算法应用到计算资源、存储容量有限的移动设备上的效果往往是不尽人意的，因此，需要针对移动设备及其作用的数据的特点，设计出合适的压缩算法。本文对现有的移动设备文件压缩研究进行了分析，并对未来发展前景广阔的各个领域和课题中存在的各种发展问题和不同的研究目标提出了见解。

关键字　移动设备；文件压缩

Review of file compression algorithms for mobile devices

HU Ke-Xin[1)]

[1)]（Department of Computer Science and Technology, Huazhong University of Science and Technology， Wuhan　430070）

Abstract　Mobile devices have become an indispensable part of People's Daily life, and massive data will be generated when people use mobile devices, which leads to the demand for efficient technologies for data storage and transmission on mobile devices. For example, using Taobao APP to purchase goods will generate a large amount of data, including browsing history, purchase history, search history, etc., which makes it a great challenge for these data to be stored in limited storage space or communicate through the limited bandwidth of the Internet. Therefore, there is increasing research on data compression and communication theory on mobile devices to address these challenges. The size of the data can be permanently or temporarily reduced by performing compression operations on the data. In digital processing, not only the size of the data is considered, but also the time is saved. Optimizing storage media and bandwidth requires efficiency and compression processes, and meeting specific goals and needs requires different compression techniques. Mobile applications show the particular mode of file access, usually involves the main written to the file and read-only file random access, mobile equipment program relies heavily on transaction data management for SQLite embedded database layer, it will produce a large number of small files cover and additional operation, high in stress seriously affect the flash-based removable storage life. However, most of the existing data compression technologies are for traditional desktop systems, and the effect of these algorithms on mobile devices with limited

computing resources and storage capacity is often unsatisfactory. Therefore, it is necessary to design appropriate compression algorithms according to the characteristics of mobile devices and their data. This paper analyzes the existing research on file compression of mobile devices, and puts forward some opinions on various development problems and different research objectives in various fields and subjects with broad development prospects in the future.

Keywords Mobile devices; File compression

1 引 言

对于我们每个人来说，包括智能手机、平板电脑和可穿戴设备在内的移动设备现在已成为日常生活中的必需品。近期研究报告称，2019 年智能手机出货量超过 13.7 亿部[1]。随着互联网技术的发展，人们使用移动设备产生的数据体量越来越大，任何一款稍微消耗性能的手游都需要 10G 起步的存储容量，如热门游戏王者荣耀与原神分别需要至少 13G 与 18G 的存储容量。当要存储的数字文件非常大，超过了剩余的存储空间时，小的存储区域就会成为一个急需解决的问题，而用于记载图像或视频文件的大量数据则需要大量的空间来存储。然而，通过增加存储容量来缓解内存的压力，例如增加硬盘，这需要相当昂贵的成本，并不总是一个好的解决方案。成本问题是缓解移动设备存储压力中不可避免的问题，在当前，价格相对较低的低端智能手机仍然在市场上占据主流地位[2,9]，尤其是在发展中国家，然而，此类设备通常配备的容量和性能资源都有限。例如，低端 Android 智能手机可能具有 1-2GB 运行时内存和 8-16GB 慢速 eMMC 存储[3,4,5]。更糟糕的是，Android 操作系统本身可以消耗超过 3 GB 的存储空间，因此，可供用户使用的存储空间更为稀缺。即使对于高端智能手机，流行或常驻应用程序的不断增加的存储和运行时内存消耗通常也会导致用户启动和系统启动操作的设备资源不足。

在使用移动设备传输文件时，文件大小也会成为一个问题，因为文件的传输速率除了受到互联网速度的影响外，还受到文件大小的影响。传输的文件越大，则处理时间越长，导致计算机网络越繁忙，而移动设备计算资源有限，因此各种类型的视频和音频文件在通过互联网发送时需要很长时间。改善传输媒体的性能可以通过增加带宽来实现，但这会导致较高的互联网成本。

此外，移动设备一般采用闪存进行持久数据存储。虽然移动处理器的性能正在大幅提升，但移动设备存储性能的提升却是相对较为缓慢。最近的研究表明，移动存储上的 I/O 操作以写为主[10-14]，并且写入模式是高度随机和同步的。由于闪存具有相对较高的写入延迟[15,16]，这些写入操作一般被认为与用户感知的延迟密切相关。此外，随着闪存技术以降低耐用性为代价向高单元位密度发展，高写入压力会对闪存寿命产生负面影响。

采用压缩技术来解决数据存储以及数据传输问题是一个可实现的选择。数据压缩是将输入数据流(源数据流或本地数据流)转换成较小的数据流(输出流、位流或压缩流)的过程。具有压缩支持的文件系统或压缩文件系统可通过在访问时透明地压缩/解压缩文件数据来向用户释放更多空间，以及加快数据传输的速度，提高用户的体验感。

传统的压缩技术大多是基于桌面系统的，而以 Android 为主流的移动设备表现出非常不同的数据文件形式与使用方式。首先大多数以移动应用程序的文件都是“读热写冷”的，即文件经常被读取而更新相对较少，有些文件甚至是只读的。因此，对这些“读热写冷”文件进行压缩是减少操作系统/应用程序存储空间的自然选择。但是，Android/iOS 下的主流文件系统均不支持这种压缩。其次，主流移动设备操作系统 Android 程序依赖于嵌入式数据库层 SQLite 进行事务数据管理，SQLite 在运行时会生成许多小文件覆盖和追加操作对于较小的追加操作，文件压缩无法对新数据使用较大的压缩窗口以获得更好的压缩结果。对于 Android 可执行文件，虽然它们是在安装时顺序写入的，但它们在应用程序启动期间会受到小的随机读取操作。由于块 I/O 的单元大小较大，因此从随机文件偏移量解压缩文件块会显着放大 I/O 读取开销[17,18]。然而，现有的 Android 移动设备的压缩方式并没有涉及到上述这些数据库文件和可执行文件的这些独特的文件访问模式。再者，现在许多被开发出来的压缩程序大多数是非常有效的读取小文件，而且效率低，并不

适用于非常大的文件，尤其体现在计算能力较弱的移动设备上。

近年来移动设备中数据存储及传输的重要性不断被人提及，而现有的压缩技术也不能很好的解决问题，因此，越来越多的研究者将目光投向了移动设备中文件的压缩研究，取得了一定的成果，本文将对近年来的研究成果进行分析整合。

2 背景和原理

移动设备的产生及使用的数据体量不断增长，然而移动设备的存储容量、传输带宽以及计算资源稀缺。现有的数据压缩技术并不完全适用于移动设备中数据的特性，因此，需要对其数据特性进行分析，探索更为合适的移动设备数据压缩技术。

2.1 用户感知存储空间不足

由于成本限制，智能手机通常资源稀缺。同时，Android 操作系统所占用的空间也在不断增加。图 1 显示了不同 Android 版本的存储容量[19]中的 /system 分区大小。Sparse Image 去掉了所有零块，因此只包含所有有效数据；而 Raw Image 是存储到设备中后消耗的实际空间。从图中可以看出/system 的数据大小分区从 Android 2.3.6 中的 184MB 增加到 Android 9.0.0 中的 1.9GB。除了有效数据量增加的趋势外，我们还可以看到 Android 7 和 8 中出现了大量的零块，它们也占用了很大的空间。对于 Android 9，零块明显更少，这是由于 ext4 文件系统中支持数据块重复数据删除[20]。除了图 1 所示的 /system 分区之外，还有其他 USENIX Association 2019 USENIX Annual Technical Conference 1189Android 的空间消耗分区，例如 /vendor、/oem 和/odm [21]。正如相关研究者报告的那样，整个 Android 系统本身使用的空间呈现增加的趋势，并且远远大于上述所展示的存储空间。例如，Android 6.0.0 在恢复出厂设置后消耗 3.17GB 存储空间。与此同时，Android 应用程序的存储空间消耗也在不断增长。据 Google Play 报道，到 2017 年初，与 Google 启动 Android 应用程序市场时相比，平均应用程序大小增加了五倍[22]。

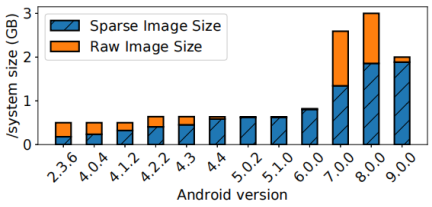

图 1 Android /system 分区大小

综上所述，对于低端手机用户而言，可供用户使用的存储容量是非常小的。此外，许多智能手机的顶级应用程序往往会消耗大量内存，如一些需要消耗内存的游戏，即使在高端智能手机上也只留下少量内存用于系统启动操作。

2.2 高写入压力

Android I/O 系统由主机系统软件和闪存芯片组成。主机软件包括一个轻量级数据库层 SQLite，用于事务数据管理。SQLite 在文件系统层之上运行，其中提供了两个主要选项：就地更新文件系统 Ext4 和 F2FS，一个日志结构的文件系统。基于 Android 操作系统的移动应用程序严重依赖 SQLite 日志机制来保证数据完整性，从而会产生大量的同步的随机块写入，而 I/O 系统的其他组件在此基础上进一步放大了写入流量。具体来说，F2FS 的空闲空间碎片整理涉及许多额外的数据迁移，移动存储中的闪存垃圾收集需要在内存擦除之前移动数据。当文件系统充满度级别高时，多级写放大将会变得更糟。放大的写入流量显着降低了用户感知的延迟。此外，随着现代闪存技术以降低耐用性为代价向高位单元密度发展，过多的写入流量还会影响到闪存芯片的使用寿命。例如，TLC 闪存芯片只能承受大约 1,000 个 P/E（编程-擦除）周期[23]。

2.3 移动设备中文件的访问特性

文件压缩的读取延迟开销取决于读取请求的大小，即读取请求越小，读取放大率越高，因而导致读取延迟开销越大。为了测量移动设备中的读取访问特性，Xuebin Zhang 等人增强了移动存储分析器以使其在 Android 4.4 下工作。因此，他们在 Android 平板电脑 Google Nexus 7（1.5 GHz Snapdragon 四核、2 GB RAM、16 GB eMMC、Android 4.4.3 内核 3.4）上收集了四个应用程序（Facebook、Instagram、Gmail 和 Angrybird）的块跟踪.0）。图 2 显示了启动 App 时的读取请求特性。他们根据文件类型将读取请求分为五组：可执行文件（.dex、.apk 和 .so）、数据库.db 和 .db-

journal 等）、多媒体（.mp3、.rmvb 等）、资源（.xml、.dat 等）等。如图 2（a）所示，可执行文件占总读取请求的 75% 以上。图 2（b）则显示了读取请求大小的分布。显然，小的读取请求占总读取流量的很大比例，例如，平均 40%的请求为 4 kB，超过 65% 的请求不大于 16 kB。

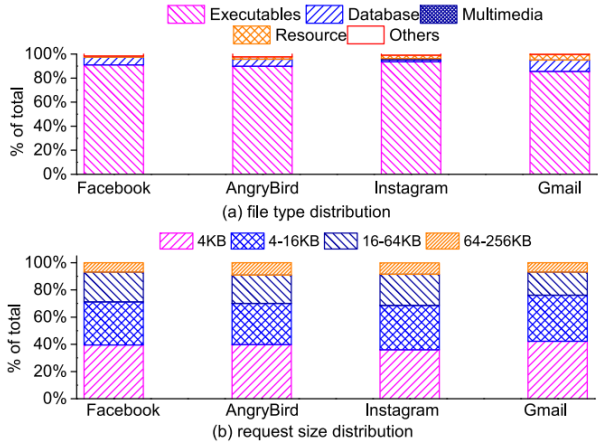

图 2　读请求分布

除了对读写文件大小的区别，移动应用程序对于不同类型的文件的访问行为也是不同的，根据访问类型（读或写）和热度（访问频率），可以将文件分为两类：第一类是写热、读冷文件，第二类是写冷读热文件。前者主要包括回滚日志 (*.db-journal)，因为它们经常被写入[24]，同时它们很少被读取，除非在崩溃恢复期间。同样，SQLite 数据库文件也属于这一类文件，因为许多移动应用程序经常编写 SQLite 数据库文件，但很少读取它们[25]。第二类文件主要包括可执行文件，它们在安装或更新后是不可变的。Android 可执行文件很大且高度可压缩。

3　研究进展

移动应用产生的文件数据越来越多，而存储设备容量的增长已经不能满足增长的数据存储需求，同时过大的数据体量也对数据传输造成一定的影响，因此，需要引入文件压缩技术来缓解存储及传输压力。为了提高移动设备用户在使用设备过程中的体验感，不同的研究学者针对移动应用程序的文件特性有着不同的侧重点，进而提出了不同的压缩算法。

3.1　只读文件压缩

移动设备中存在着许多的“读热写冷”文件，即文件经常被读取而更新相对较少，且有许多文件甚至是只读的，因此压缩只读文件系统可以显着减少只读系统资源使用的存储空间。

3.1.1　Squashfs

Squashfs[26]是 Linux 中广泛使用的压缩只读文件系统，它具有许多功以及中等的性能。它支持多种压缩算法，chunk（即压缩输入）大小可以在 4KB 到 1MB 之间选择。在 Squashfs 中，元数据可以被压缩，并且 inode 和目录的存储更加紧凑。文件数据逐块压缩，压缩后的数据块按顺序存储。每个原始数据块的压缩大小存储在 inode 内的列表中。这些大小用于在解压缩过程中定位压缩块的位置。然而，Squashfs 并不适用于资源稀缺的移动设备，因为它会导致致性能和内存消耗的显着开销。

Xiang Gao[17] 等人尝试将 Squashfs 用于 Android 上的只读分区，发现虽然系统使用 Squashfs 成功启动，但即使背景工作负载较轻，启动相机应用程序也需要数十秒的时间。造成 Squashfs 在 Android 上的性能降低的原因有两个，第一个是 I/O 放大，他们使用 FIO 算法[27]来评估 Squashfs 的基本性能，当 Android 从 Squashfs 中存储的 1GB enwik9[28]文件中顺序读取 16MB 时，实际发出的 I/O 大小为 7.25MB。虽然关于压缩的数字看起来不错，但 Squashfs 问题 Android 随机读取 16MB 时需要造成 165.27MB I/O 读取。而且，当 Android 读取每 128KB 的前 4KB 时，读取 16M 的文件数据发出多达 203.91MB 的 I/O 读取。差异表明，当 Squashfs 读取一些之前没有解压和缓存的数据时，请求的数据大小被显着放大。第二个原因是额外的内存消耗。在 Squashfs 上顺序读取 1GB enwik9 文件后的总内存消耗约为 1.35GB，这表明与所需的原始数据大小相比，Squashfs 中的解压缩需要大量的临时内存。这给 Android 带来了很大压力，因为鉴于 Android 及其应用程序已经消耗了大量内存，因此内存是用户体验的关键因素。一方面，在解压过程中分配内存可能会触发内存交换，这涉及到牺牲选择和高成本的 I/O。另一方面，在解压缩期间消耗大量额外内存会通过丢弃缓存数据或换出有用的内存页面影响其他组件或应用程序。Squashfs 的上述两个缺陷揭示了在为资源稀

缺的智能手机设计压缩只读文件系统时面临的两个挑战，即不牺牲压缩率的情况下减少解压过程中的 I/O 放大，以及减少解压过程中的内存消耗以防止性能下降。

3.1.2 EROFS

为了节省存储空间并以低内存开销保持高性能，针对 Squashfs 只读压缩系统的缺陷问题，Xiang Gao[17]等人设计并实现了 EROFS，一种具有压缩支持的增强型只读文件系统。EROFS 引入固定大小输出压缩，将文件数据压缩成多个固定大小块，以显着减轻读取放大问题并尽可能减少不必要的计算。EROFS 利用压缩算法（如 LZ4）的特点，设计了不同的内存高效解压方案，以减少解压过程中额外的内存使用。他们设计的 EROFS 具有固定大小的输出压缩、缓存的 I/O 和就地 I/O。

EROFS 使用滑动窗口压缩文件数据来获得固定大小的输出压缩，滑动窗口的大小为固定值并且可以在生成图像时进行配置。压缩算法被多次调用，直到所有文件数据被压缩。例如，对于 1MB 的滑动窗口，EROFS 向压缩算法提供 1MB 的原始数据一次数据。然后，该算法尽可能地压缩原始数据，直到所有 1MB 的数据被消耗或消耗的数据可以生成正好 4KB 的压缩数据。剩下的原始数据与更多的数据相结合，形成另一个 1MB 的原始数据，用于下次调用的压缩。

在实际解压缩之前，EROFS 需要空间来存储从存储器中检索到的压缩数据。而这对于固定大小的输入压缩来说代价很高，因为内存分配过多，甚至是固定大小的页面交换。由于 EROFS 清楚地知道每个压缩只能检索到两个压缩块。EROFS 有两种策略:缓存 I/O 和就地 I/O。EROFS 使用缓存的 I/O 将部分解压的压缩块。EROFS 管理一个特殊的 inode，它的页面缓存存储在 inode 中按物理块号索引的压缩块。因此，对于缓存的 I/O, EROFS 将在特殊的 inode 页面缓存来发起 I/O 请求，这样压缩的数据将直接被取到分配的数据中页存储驱动程序。对于将被完全解压缩的压缩块，EROFS 使用就地 I/O。在读取每个文件时，VFS 将在页面缓存中为文件系统分配页面以存放文件数据。这些没有任何意义的页面解压前的数据，称之为可重用页面。对于就地 I/O, EROFS 使用最后一个可重用页面来初始化 I/O 请求。

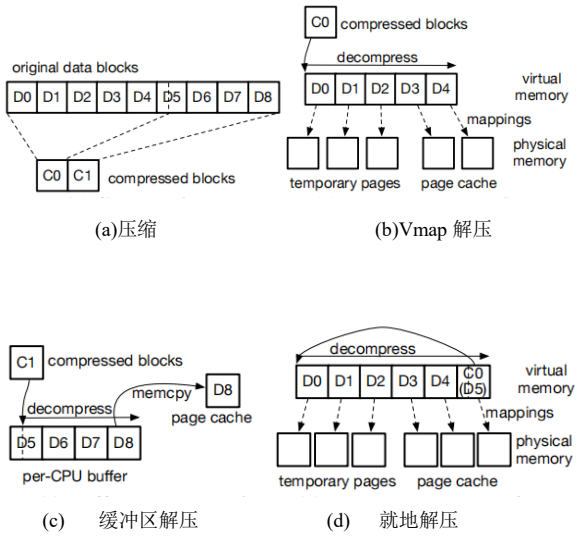

图 3 解压缩

EROFS 可以快速和高效地解压缩数据，如图 3(a)，其中前五个块（D0 到 D4）和部分块 D5 被压缩到块 C0，其余块被压缩到块 C1。为了得到块 D3 和 D4 中的数据，EROFS 首先从 storage 中读取压缩后的块 C0，存入内存。然后 EROFS 将按照以下步骤对其进行解压缩。

1. 找到压缩块（C0）中存储的最大所需块号，即示例中的第五个块（D4）。作为一个优势，EROFS 只需要解压前五个块（D0 到 D4），而不是解压所有原始数据块。

2. 对每个需要解压的数据块，找到内存空间来存放。在图 3(b)所示的例子中，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 VFS 在页缓存中分配的两个物理页。

3. 由于解压算法需要 continuousmemory 作为解压的目的地，EROFS 通过 vmap 接口将上一步准备好的物理页映射到一个连续的虚拟内存区域。

4. 如果是 in-place I/O，这种情况下压缩块(C0)存放在 page cache page 中，EROFS 还需要将压缩数据(C0)复制到一个临时的 perCPU page 中，这样解压后的数据就会被取走'在解压过程中不要覆盖压缩数据。

5. 最后调用解压算法，将压缩块中的数据提取到连续存储区。压缩后的三个临时物理页和虚拟内存区域可以被回收，并且请求的数据已经写入相应的页缓存页。

EROFS 使用 LZ4 作为压缩算法，需要回溯不超过 64KB 的解压数据。因此，对于提取超过 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。

他们将 EROFS 实现为 Linux 文件系统，并将 EROFS 的公共部分上传到 Linux 内核。实验结果表明，相较于现有的压缩只读文件系统，EROFS 在各种微基准测试中优于现有的压缩只读文件系统，并将实际应用程序的启动时间减少了 22.9%，同时几乎将存储使用量减半。

3.1.3 零延迟的 Transparent Compression

由于相关的延迟开销 Android/iOS 下的主流文件系统均不支持针对“读热写冷”文件的 Transparent Compression。其读取延迟开销来自两个方面：

（1） 访问任何 4 kB 页面都会触发从底层存储设备读取多个 4 kB 扇区，从而导致读取放大。因为它可能需要 50100 毫秒到读一个闪存物理页，读放大会引起明显的延迟开销。

（2） 数据解压会消耗 CPU 周期和额外的 CPU-DRAM 数据流量，导致额外的读取延迟开销。此外，文件系统元数据管理变得更加复杂。

针对 Transparent Compression 的读放大和解压延迟问题，Xuebin Zhang[29]等人提出了简单而有效的设计解决方案，以消除文件系统级别的读取放大并消除计算机体系结构级别的计算延迟开销。他们采用了两种简单的策略：

（1） 约束压缩单独压缩每个 4 kB 页面，并强制每个压缩的 4 kB 页面完全驻留在单个 4 kB 扇区中；

（2） 压缩感知逻辑块寻址 (LBA) 扩展，确保对现有文件系统（例如 Android 中的 ext4）进行最少修改以支持约束压缩。

为了消除零放大，他们选择单独压缩每个 4 kB 页面，并将每个压缩的 4 kB 页面完全放入单个 4 kB 扇区中，这称为约束压缩。同时，为了简化文件系统元数据数据结构和读取过程，应该满足两个要求：

（1）每个 4 kB 页面（从 OS/Apps 的角度来看）都有一个唯一的地址记录在文件系统元数据中，无论其可压缩性如何;

（2） 在确定要发送到存储设备的 LBA 时，文件系统只需要查找其元数据一次。

他们提出了一种称为压缩感知 LBA 扩展的简单方法来实现这两个要求，即设置每个 4 kB 扇区最多可以存储 2^m 个 压缩的 4 kB 页，其中 m 是一个小整数（例如，1 或 2）。假设存储设备为每个 4 kB 扇区使用一个 n 位 LBA。文件系统为每个 4 kB 页面分配一个唯一的（m+n）位扩展 LBA (eLBA)，由 n 位 LBA 和 m 位位置索引组成。该 n 位 LBA 点的扇区存储压缩的版本 4 KB 页，而 m 位位置索引指示其需要的页面。当输出缓冲区接收到 4k 字节时，解压缩器将确定压缩页面的结束。

为了消除解压延迟开销，他们在计算机架构级别提出了一个简单的设计解决方案。当应用处理器中的 I/O 控制器从存储设备中取出一个 4kB 的扇区时，它首先缓冲数据，执行数据完整性检查。一旦它接收到整个 4 kB 扇区并验证其完整性，I/O 控制器就会通过 DMA 将 4 kB 扇区数据传输到 DRAM 控制器，如图 4(a)所示。为了消除数据解压延迟，将解压引擎集成到 I/O 控制器中，如图 4(b)所示。利用解压缩的字节串行处理特性，引擎可以即时解压缩数据，而不会造成额外的延迟。由于每个 4 kB 扇区最多可包含 2^m 压缩的 4 kB 页，因此需要将 I/O 控制器内的缓冲区大小扩大 2^m 倍。一旦 I/O 控制器接收到整个 4 kB 扇区的最后一个字节，CRC 校验和数据解压缩就可以立即完成。

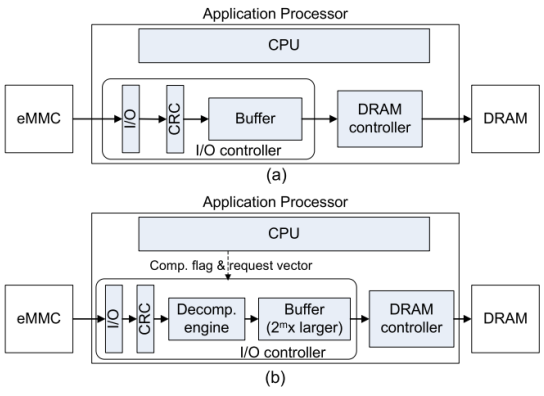

图 4 应用处理器中的 I/O 控制器结构图解

根据以上设计，他们使用 FUSE 框架实现了一个原型文件系统来进行实验，实验结果表明，在安装了 Android 5.0 的 Nexus 7 平板电脑上，操作系统/应用程序的占用空间最多可以减少 39%。且通过专用集成电路 (ASIC) 综合，他们所提出的计算机架构级设计解决方案可以以非常小的硅成本消除解压缩延迟开销。

3.2 日志结构文件系统压缩

基于 Android 的移动设备严重依赖 SQLite 进行事务管理，会生成许多小文件覆盖和追加操作，这些小操作在存储空间中高度碎片化。针对小文件追加操作，Cheng Ji 等人[30]发现，现有的顺序压缩方法出乎意料地降低了空间利用率，并放大了在移动设备上启动应用程序的读取开销，以及与传统的就地更新文件系统（例如 Ext4）相比，LFS 对文件压缩非常友好。基于以上研究内容，他们提出了一种 FPC 方式。FPC 具有前景压缩和背景压缩，前台压缩仅应用于写入密集、高度可压缩的 SQLite 文件，因为 SQLite 日志文件几乎不被读取（预写日志日志）或从不读取（回滚日志）。后台压缩方法用于识别和重新组织可执行文件中的读取临界块，以快速启动应用程序并节省空间。他们所提出的 FPC 通过在日志结构文件系统（例如 F2FS）之上的实现来实现用于移动存储。FPC 分别利用 F2FS 中现有机制的异地更新和反向映射进行前台压缩和临界块重组。

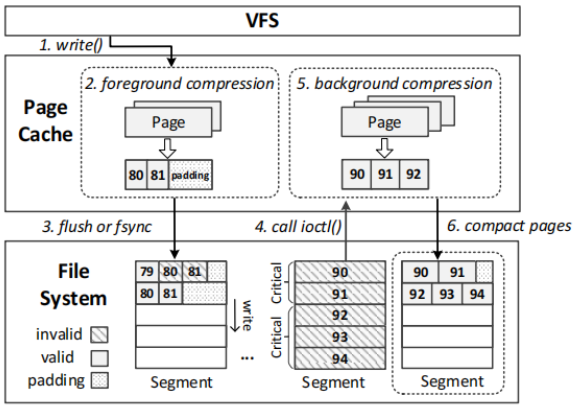

图 5 FPC 架构

对于前台压缩（FC），具有三个特点，即非顺序文件块压缩、选择性前台压缩以及元数据级文件压缩。

顺序压缩方式会导致存储空间利用率低，因为主要的写入流量贡献者 SQLite 产生许多绑定随机文件偏移量的小写入。在这种情况下，由于与先前压缩文件块无关的文件偏移量，压缩文件块通常需要新的物理块，造成严重的空间浪费。Cheng Ji 等人提出的 FC 方法采用了非顺序的压缩方式，即允许不相关文件偏移的文件块共享相同的物理块。

FC 采用了选择性文件压缩方式而不是无条件压缩，避免了多余的延迟与能耗。FC 可以简单地忽略与已知不可压缩的文件类型相关的写入，例如，具有多媒体扩展名 *.jpg、*.mp4 等的文件。同样的，FC 采用了一种采样技术来快速识别 SQLite 文件中不可压缩的内容：FC 始终压缩 SQLite 文件写入操作的第一个缓存文件页面。

对于元数据级的文件压缩，FC 将写入 *.db-journal 文件的数据块和它们关联的包含元数据信息的节点块通过压缩到同一段。同时，向每个混合块插入一
个校验和，如果在崩溃恢复过程中在混合块上检测到校验和失败，则混合块中的压缩节点块将被丢弃。

对顺序写入、随机读取的可执行文件进行前台压缩有降低应用程序启动性能的风险。这部分研究了可执行文件的访问模式并提出了后台压缩（BC）。他们通过分析了 Facebook、Chrome 和 Messenger 的主要可执行文件 base.apk 文件中读取地址的分布，发现被检查应用程序的读取请求很小且是随机的。因此他们
设计的 BC 方法中对这些读取的数据进行压缩。具体操作是，在应用程序启动期间监控读取关键数据集。稍后，根据请求，文件系统将这些关键数据压缩并压缩成文件块。

读/写模式	文件类型	压缩策略	压缩式
写密集型，随机写	D b	非序列块上的压缩	FC 可减少写入压力
（几乎）只写	db-日志	大压缩窗口，元数据级压缩	
一次写入，随机读取	apk	关键数据压缩和压缩	
一次写入，顺序读取	dex, odex	跨物理块边界的大压缩窗口	BC 用于快速启动和节省空间

图 6 不同类型文件及其读/写模式的压缩策略

基于上述思想，他们在日志文件系统 F2FS 中实现了 FPC。利用动态压缩窗口机制进行压缩，即压缩窗口的默认大小设置为 4 KB，以避免放大读取开销。例外情况如下：首先，对于 SQLite 日志的前台压缩，压缩窗口随着块写入请求而变大。因为这些文件几乎没有被读取，所以使用大的压缩窗口对读取性能影响不大。其次，后台压缩使用 32 页 (128 KB) 作为 *.dex 和 *.odex 的压缩窗口大小。利用子块 L2P 映射映射机制来提高读写数据压缩功效，利用反向映射机制进行解压缩，以及通过继承 F2FS 的 6 个记录头并增加三个新的日志记录头来进行数据分离。

通过在一组流行的移动应用程序上进行实验评估，实验结果表明，对于他们所提出的 FPC 方案，可以使 SQLite 写入量的减少导致整个系统写入量减少了 26.1%，进而减少闪存磨损程度提高闪存寿命。在写入延迟方面，FPC 相对于没有任何压缩方式的 F2FS 而言，将 SQLite 文件的写入延迟平均降低了 7.1%。在能耗方面，FPC 在大多数应用中都实现了较低的能耗。在存储空间方面，FPC 使所有应用程序的总可执行文件大小从 846 MB 显著减少到 646 MB（减少了 23.7%）。在应用程序启动时间方面，FPC 也减少了这些应用程序的启动时间，平均减少了 5.2%。

然而对于设备级的压缩而言，无条件的设备端数据压缩并没有意识到很多有用的主机信息，例如与可执行文件相关的关键读取，因此很难优化解压缩延迟并改善用户体验。设备压缩的另一个关键问题是最近的 Android 版本已经配备了块加密，这使得加密数据无法压缩。

4 总结和展望

移动设备的不断普及以及移动应用程序的不断发展，导致了移动程序数据规模的不断扩大，为新的适用于存储资源紧缺即计算能力较弱的移动设备的文件压缩技术的发展提供了持久的动力。综合国内外研究现状可以看出，移动设备文件具有高的写入压力，以及小的碎片化的随机化的读写请求。移动设备运行时的运行资源可以打包成大型可执行文件，这些文件大多是“读热写冷”的，甚至是只读的，同时这些文件也是高度可压缩的。针对这些表现出与传统桌面系统文件不同特性的数据文件，传统的压缩方式已经不能提供更好的压缩性能了，因此需要探索更为合适的压缩方式。

结合上述文件特点，近年来有许多研究者提出了不同的压缩设计方案。

针对可执行文件“读热写冷”以及只读的特点，Xiang Gao[17]等人与 Xuebin Zhang 等人[29]提出了各自的文件压缩方式。前者提出了一种具有压缩支持的增强型只读文件系统 EROFS，通过具有固定大小的输出压缩、缓存的 I/O 和就地 I/O 的设计方案来获得更好的压缩性能。后者设计了一种零延时的 Transparent Compression 方式，通过实现一种称为压缩感知 LBA 扩展的方法来消除读放大，以及一种基于计算机架构的方法来消除解压延迟，从而获得更好的压缩性能。

在对“读热写冷”的可执行文件的压缩基础上，结合 SQLite 的“写热读冷”文件，Cheng Ji 等人[30]提出了一种 FPC 压缩方法，将对于前者的背景压缩与对于后者的前台压缩结合起来，进一步减少用户启动程序的时间以及存储空间的利用率。

然而，现有的研究仍然具有不足之处，针对只读文件的压缩设计并没有考虑到移动设备中仍然存在着许多“写热”的文件，具有一定的局限性，而 Cheng Ji 等人[30]提出的算法对于设备级的压缩而言具有局限性。总而言之，对于移动设备文件的压缩方式的探索仍然具有很大的研究空间。

[1]Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/, 2018.

[2] Android one was conceived with india in mind, says Google’s Sundar Pichai. https://gadgets.ndtv.com/mobiles/news/googles-sundar-pichai-on-android-one-in-an-exclusive-chat-withndtvs-vikram-chandra-592062, 2014.

[3] HUAWEI Y3 2018. https://consumer.huawei.com/za/phones/y3-2018/specs/, 2018.

[4] Libext2fs: add EXT2_FLAG_SHARE_DUP to de-duplicate data blocks. https://android-review.googlesource.com/c/platform/external/e2fsprogs/+/642333, 2018.

[5] Nokia 2.1 - long lasting entertainment. https://www. nokia.com/phones/en_int/nokia-2, 2018.

[6] Samsung unveils the Galaxy J2 core; an introductory smartphone packed with performance. https://news. samsung.com/global/samsung-unveils-the-galaxy-j2-core-an-introductory-smartphone-packed-with-performance, 2018.

[7] AXBOE, J. Flexible I/O tester. https://github.com/axboe/fio.

[8] BENAVIDES, T., TREON, J., HULBERT, J., AND CHANG, W. The enabling of an Execute-In-Place architecture to reduce the embedded system memory footprint and boot time. *JCP 3*, 1 (2008), 79–89.

[9] BRUMLEY, J. Apple, Samsung continue to lose smartphone market share in shift toward more value. https://seekingalpha.com/article/4101007- apple-samsung-continue-lose-smartphone-market-share-shift-toward-value, 2017.

[10]Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4), 2012.

[11] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: elim-inating the fifilesystem journaling in resolving the journaling of journal-anomaly.In *Proceedings of USENIX ATC*, pages 235–247, 2015.

[12] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Jason Chun Xue. Inspection and characterization of app fifile usage in mobile devices. *ACM Transactions on Storage (TOS)*, 16(4), 2020.

[13] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of ATC, 2013*, pages 309–320, 2013.

[14] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Reliable, consistent, and effificient data sync for mobile apps. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 15)*, pages 359–372, 2015.

[15] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving fifile system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of Annual Technical Conference (USENIX ATC 17)*, pages 759–771. USENIX Association, 2017.

[16] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. Fasttrack: Foreground app-aware I/O management for improving user experience of android smartphones. In *Proceedings of Annual Technical Conference (USENIX ATC 18)*, pages 15–28. USENIX Association, 2018.

[17] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. EROFS: A compression-friendly readonly fifile system for resource-scarce device. In *Proceedings of Annual Technical Conference (USENIX ATC)*. USENIX Association, 2019.

[18] Xuebin Zhang, Jiangpeng Li, Hao Wang, Danni Xiong, Jerry Qu, Hyunsuk Shin, Jung Pill Kim, and Tong Zhang. Realizing transparent os/apps compression in mobile devices at zero latency overhead. *IEEE Transactions on Computers*, 66(7):1188–1199, 2017.

[19] Factory images for Nexus and Pixel devices. https:// developers.google.com/android/images.

[20]Libext2fs: add EXT2_FLAG_SHARE_DUP to deduplicate data blocks. https://android-review. googlesource.com/c/platform/external/ e2fsprogs/+/642333, 2018.

[21] Partitions and images. https://source.android. com/devices/bootloader/partitions-images.

[22] TOLOMEI, S. Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinkingapks-growing-installs-5d3fcba23ce2.

[23] Samsung Semiconductors. 3D TLC NAND to beat MLC as top flflash storage. EETimes, 2015.

[24] M. Son, J. Ahn, and S. Yoo. Nonvolatile write bufferbased journaling bypass for storage write reduction in mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1747– 1759, 2018. USENIX Association 19th USENIX Conference on File and Storage Technologies 139

[25] Taeho Hwang, Myungsik Kim, Seongjin Lee, and Youjip Won. On the I/O characteristics of the mobile web browsers. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18)*, pages 964– 966, 2018.

[26] SQUASHFS 4.0 fifilesystem. https://www.kernel. org/doc/Documentation/filesystems/squashfs. txt.

[27] AXBOE, J. Flexible I/O tester. https://github.com/axboe/fio.

[28] MAHONEY, M. About the test data. http://mattmahoney.net/dc/textdata.html, 2011.
[29] Xuebin Zhang, Jiangpeng Li, Hao Wang, Danni Xiong, Jerry Qu, Hyunsuk Shin,
Jung Pill Kim, Tong Zhang. Realizing Transparent OS/Apps Compression in Mobile Devices at Zero Latency Overhead. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 66(7):1188-1199, 2017
[30] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, Tei-Wei Kuo, Chun Jason Xue. Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices, In *Proceedings of Annual Technical Conference (USENIX ATC)*. USENIX Association, 2021