乐观的崩溃一致性

Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, 安德列

威斯康星大学计算机科学系，
麦迪逊

{Vijayc, MaDaNuu, DuSauu, ReMiZ} C.WISC EDU

摘要

我们介绍了乐观的崩溃一致性，这是日记文件系统中崩溃一致性的一种新方法。通过使用一系列新颖的技术，我们演示了如何构建乐观的提交协议，该协议可以从崩溃中正确恢复并提供高性能。我们在称为 extFS 的 Linux ext4 变体中实现了这种乐观方法。我们引入了两个新的文件系统原语 osync（）和 dsync（），它们使写入的顺序与持久性脱钩。我们通过实验表明，OptFS 可以提高许多工作负载的性能，有时可以提高一个数量级。我们通过一系列的稳健性测试确认了它的正确性，表明它在崩溃后恢复到一致的状态。最后，我们证明 osync（）和 dsync（）在原子文件系统和数据库更新方案中很有用，既可以提高性能，又可以满足应用程序级的一致性要求。

1 介绍

现代存储设备为客户端提供了看似无害的界面。要读取一个块，只需发出一个低级读取命令并指定要读取的块（或一组块）的地址即可；磁盘完成读取后，会将其转移到内存中，所有等待完成的客户端都将收到通知。写操作遵循类似的过程。

不幸的是，在现代磁盘中引入写缓冲[28]大大简化了这个看似简单的过程。启用写入缓冲后，磁盘写入可能会乱序完成，因为智能磁盘调度程序可能会重新排序对性能的请求[13、24、38]；此外，写问题后收到的通知仅暗示磁盘已收到请求，而不是数据已被持久地写入磁盘表面。

乱序的写入完成尤其使从系统崩溃中恢复的已知技术变得非常复杂。例如，诸如 Linux ext3，XFS 和 NTFS 之类的现代日志文件系统都仔细地精心安排了一系列更新，以确保对主文件系统结构和日志的写操作以特定顺序到达磁盘[22]。在更新某些结构时，LFS，btrfs 和 ZFS 等复制写入文件系统也需要排序。如果没有命令，大多数文件系统无法确保崩溃后可以恢复状态[6]。

现代驱动器中通过昂贵的缓存刷新操作来实现写入顺序[30]；这样的刷新会导致驱动器中所有缓冲的脏数据立即写入表面（即，持久存在）。为了确保 A 在 B 之前写入十，客户端向 A 发出写操作，然后进行缓存刷新；当刷新返回时，客户端可以安全地假定 A 已到达磁盘；然后可以安全地发出对 B 的写操作，因为知道在 A 之后将继续存在。

不幸的是，缓存刷新很昂贵，有时甚至是昂贵的。刷新会使 I / O 调度效率降低，因为磁盘具有较少的选择请求。刷新也不必要地将以前的所有写操作强制写入磁盘，而客户端的要求可能不太严格。另外，在大的高速缓存刷新期间，磁盘读取可能会等待很长的等待写入，从而表现出非常长的等待时间[26]。最后，冲洗破坏了顺序和耐用性。如果客户只是希望先订购一个写，那么将第一次写强制到磁盘是达到这种目的的昂贵方法。简而言之，冲洗的经典方法是悲观的。它假定将发生崩溃，并且会尽力确保通过刷新命令磁盘永远不会处于不一致的状态。悲观主义导致的性能不佳导致某些系统无法使用冲洗功能，显然会牺牲性能的正确性。例如，Linux ext3 默认配置多年没有刷新缓存[8]。

禁用刷新并不一定会导致文件系统不一致，而是会将其引入。我们将这种方法称为概率崩溃一致性，其中崩溃可能会导致文件系统不一致，这取决于许多因素，包括工作负载，系统和磁盘参数，以及崩溃的确切时间。

SOSP'13，11 月。2013 年 3 月 6 日至 6 日，美国宾夕法尼亚州法明顿。ACM 978-1-4503-2388-8 / 13/11。

http://dx.doi.org/10.1145/2517349.2522726

崩溃或断电。在本文中，我们的第一个贡献是对概率崩溃一致性的仔细研究，其中我们显示了哪些确切因素会影响崩溃使文件系统不一致的可能性（第 3 节）。我们表明，对于某些工作负载，概率方法很少会导致文件系统不一致。

不幸的是，对于需要确定崩溃恢复的许多应用来说，概率方法是不够的。确实，对于某些工作负载，我们还表明不一致的可能性很高。为了实现更高级别的应用程序级别的一致性（即，DBMS 可能需要的东西），文件系统必须提供的不仅仅是概率和机会。因此，在本文中，我们介绍了乐观的崩溃一致性，这是一种构建崩溃一致的日记文件系统的新方法（第 4 节）。这种乐观方法利用了以下事实：在许多情况下，可以通过其他方式来实现排序，并且崩溃是罕见的事件（类似于乐观并发控制[12，16]）。但是，以乐观的方式实现一致性并非没有挑战。因此，我们开发了一系列新颖的技术，包括对事务校验和的新扩展[23]，以检测数据/元数据不一致，延迟重用块以避免错误的悬空指针，以及选择性数据日志技术来正确处理块覆盖。这些技术的结合带来了高性能和确定性一致性。在极少数情况下确实发生崩溃，乐观的崩溃一致性既可以避免设计方面的不一致，也可以确保磁盘上存在足够的信息以在恢复过程中检测和丢弃不正确的更新。

我们通过乐观文件系统（OptFS）的设计，实现和分析，展示了乐观崩溃一致性的强大功能。OptFS 基于乐观崩溃一致性的原则来实施乐观日志记录，从而确保即使发生崩溃，文件系统也保持一致。乐观日志记录是对 Linux ext4 文件系统的一组修改，但也需要在磁盘接口上稍作更改，以提供我们所谓的异步持久性通知，即，除了在什么情况下还可以在保持写操作时发出通知。磁盘只是接收到写入。我们描述了实现的详细信息（第 5 节）并研究了其性能（第 6 节），表明 OptFS 在悲观的日志记录下明显优于经典 Linux ext4，并且在概率方面与 linux ext4 几乎相同。日志记录，同时确保崩溃一致性。

OptFS 性能优势的核心是订购和耐用性的分离。通过允许应用程序在不引起磁盘刷新的情况下订购写操作，并在需要时请求持久性，OptFS 实现高性能的应用程序级一致性。OptFS 引入了两个新的文件系统原语：osync（）和 dsync（），它们确保写操作之间的顺序，但只能保证最终的持久性，而 dsync（）则确保立即的持久性和顺序。

我们将展示这些原语如何为构建更高级别的应用程序一致性语义（§7）提供有用的基础。具体来说，我们展示了文档编辑应用程序如何使用 osync（）来实现文件的原子更新（通过创建，然后通过原子重命名），以及 SQLite 数据库管理系统如何使用文件系统提供的排序来实现有序事务。具有最终的耐用性。我们证明了这些原语足以在高性能下实现有用的应用程序级一致性。

当然，乐观方法虽然在许多情况下有用，但并不是万灵药。如果应用程序需要立即的同步持久性（而不是最终的异步持久性和一致的顺序），则仍然需要昂贵的缓存刷新。在这种情况下，应用程序可以使用 dsync（）来请求持久性（以及排序）。但是，通过将写入的持久性从其顺序中分离出来，OptFS 提供了有用的中间立场，从而为许多应用程序实现了高性能和有意义的崩溃一致性。

2 悲观的崩溃一致性

为了了解日志记录的乐观方法，我们首先描述日志文件系统中的标准悲观崩溃一致性。为此，我们描述了必要的磁盘支持（即，缓存刷新命令）以及有关此类崩溃一致性如何运行的详细信息。然后，我们演示了悲观日志记录期间缓存刷新对性能的负面影响。

2.1 磁盘接口

为了便于讨论，我们假设存在磁盘级缓存刷新命令。在 ATA 系列驱动器中，这称为“刷新缓存”命令。在 SCSI 驱动器中，它被称为“同步缓存”。两种操作都具有相似的语义，强制将磁盘中所有未决的脏写写入表面。注意，刷新可以作为单独的请求发出，也可以作为对给定块 D 写入的一部分发出；在后一种情况下，未完成的写入在写入 D 之前被刷新。

还存在一些更细粒度的控件。例如，“强制单元访问”（FUA）命令可完全读取或写入高速缓存。FUA 通常与刷新命令一起使用。例如，要在 B 之前写 A，但又要确保 A 和 B 都是持久的，客户端可以写 A，然后在启用缓存刷新和 FUA 的情况下写 B；这样可以确保当 B 到达驱动器时，A（和其他脏数据）将被强制插入磁盘；随后，由于 FUA，B 将被迫插入磁盘。

2.2 悲观日记

给定上述磁盘接口，我们现在描述日记文件系统如何安全地将数据提交到磁盘，以便在系统崩溃时保持一致性。我们的讨论基于有序模式的 Linux ext3 和 ext4 [34、35]，尽管我们所说的大部分内容都适用于其他日志文件系统，例如 SGI XFS [32]，Windows NTFS [27]和 IBM JFS [3]。]。

在有序模式下，记录文件系统元数据以保持其一致性。不会记录数据，因为两次写入每个数据块会大大降低性能。当应用程序更新文件系统状态时，元数据，用户数据或（通常）两者都需要以持久方式进行更新。例如，当用户将一个块附加到文件时，必须将新的数据块（D）写入磁盘（有时）；此外，

还必须更新各种元数据（M），包括文件的 inode 和位图将块标记为已分配。

我们将元数据的原子更新称为事务。在将事务 Tx 提交到日志之前，文件系统首先写入所有数据块（D）与交易相关的最终目的地；在事务提交之前写入数据可确保已提交的元数据不会指向垃圾。这些数据写入完成后，文件系统将使用日志记录元数据更新。我们将这些期刊文章称为 JM。保留这些写操作后，文件系统将写操作提交到提交块（JC）。当磁盘保留 JC 时，事务 Tx 被称为已提交。最后，在提交之后，文件系统可以自由更新就位的元数据块（M）；如果在此检查点过程中发生崩溃，则文件系统可以简单地通过扫描日志并重播已提交的事务来恢复。详细信息可以在其他地方找到[22，35]。

因此，我们必须进行以下一组有序写入：D 在 JM 之前，JC 之前，M 在 J 之前，或者更简单地说：D→JM→JC→M。请注意，D，JM，和 M 可以代表一个以上的块（更大交易），而 JC 始终是单个扇区（出于写入原子性的考虑）。为了实现这种排序，文件系统会在需要排序的任何地方（即在有→符号的地方）发出缓存刷新。

已建议在此协议中进行优化文献，其中一些已在 Linux ext4 中实现。例如，有些人注意到数据和日记的元数据（D→JM）之间的顺序是多余的。删除订单有时可以改善性能（D | JM→JC→M）[22]。

其他人则建议进行“交易检查-sum”[23]，可用于删除日记元数据和日记提交（JM 和 JC）之间的顺序。在正常情况下，文件系统无法一起发布 JM 和 JC，因为驱动器可能会对其重新排序。在这种情况下，JC 可能会先命中磁盘，此时系统临时崩溃（或断电）将使该事务处于看似已提交的状态，但带有垃圾内容。通过计算整个事务的校验和并将其值放在 JC 中，可以一起执行对 JM 和 JC 的写操作，从而提高性能。使用校验和时，崩溃恢复可以避免重放未正确提交的事务。通过这种优化，顺序为 D→JM | JC→M（其中日志更新上的横条表示通过校验和进行保护）。

有趣的是，这两个优化未合并，即 D | JM | JC→M 不正确；如果文件系统一起发布 D，JM 和 JC，则 JM 和 JC 可能首先到达磁盘。在这种情况下，元数据在数据之前提交；如果在写入数据之前发生崩溃，则提交的事务中的 inode（或间接块）可能最终指向垃圾数据。

奇怪的是，ext4 通过“正确”的一组安装选项允许这种情况。

我们应该注意，更新之间还存在另一个重要的顺序，特别是事务之间的顺序。日志文件系统假定事务按顺序提交到磁盘（即 Txi→Txi + 1）[35]。不遵循此顺序可能会导致在崩溃恢复。例如，块 B 本可以在 Txi 中释放，然后在 Txi + 1 中重用；在这种情况下，在提交 Txi + 1 之后但在 Txi 提交之前的崩溃将导致状态 B 分配给两个文件。

最后，最重要的是，我们提请注意此方法的悲观性质。每当需要排序时，都会发出昂贵的高速缓存刷新，从而在可能仅需要刷新其中一部分的情况下，将所有挂起的写入都强制写入磁盘。另外，即使写入可能以正确的顺序写入磁盘，也将发出刷新，具体取决于调度，工作负载和其他详细信息。冲洗会导致不必要的额外工作。最后，也许是最有害的是，尽管很少发生撞车事故，但仍增加了冲水的负担，因此在预计到极少数偶然事件的情况下会付出沉重的成本。

2.3 冲洗性能影响

为了更好地了解悲观日志记录期间缓存刷新对性能的影响，我们现在执行一个简单的实验。具体来说，我们在 Linux ext4 上运行 Varmail 基准测试，无论是否进行缓存刷新；启用缓存刷新后，我们还启用事务校验和以查看其性能影响。Varmail 在这里是一个不错的选择，因为它可以模拟电子邮件服务器，并且包括许多对磁盘的小型同步更新，因此对上述日记机制造成了压力。第 6 节中将详细介绍此基准测试和配置的实验设置。

从图 1 中，我们可以观察到以下内容。首先，交易校验和会稍微提高性能，

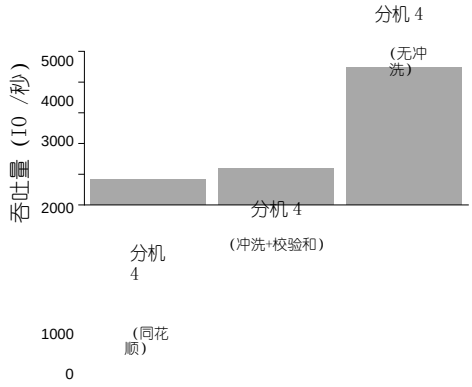

图 *1：冲洗成本。该图显示了*

在不同的 ext4 配置上 Filebench Varmail 的性能。禁用冲洗后，性能提高 5 倍。

显示删除单个订购点（通过校验和）如何有帮助。其次，也是最重要的一点是，当关闭缓存刷新功能时，性能有了很大的提高，在这种情况下几乎提高了五倍。鉴于这种巨大的性能差异，在某些安装中，禁用了缓存刷新功能，这引发了以下问题：禁用刷新功能时，会提供哪种崩溃一致性？令人惊讶的是，正如我们现在所描述的，答案不是“无”。

3 概率崩溃一致性

考虑到潜在的性能提升，从业人员有时会放弃发布冲洗并选择禁用冲洗的正确实现所提供的安全性[8]。在这种快速模式下，存在文件系统不一致的风险。如果崩溃发生在更新序列中的不及时点，并且块已在排序点之间重新排序，则文件系统运行的崩溃恢复将导致文件系统不一致。

在某些情况下，从业人员观察到，尽管存在（偶尔）崩溃，有时跳过刷新命令有时也不会导致可观察到的不一致。这样的评论引发了 Linux 社区内部有关根本原因的辩论。长期的内核开发人员 Theodore Ts’o假设了为什么尽管文件系统缺乏命令强制执行却经常实现这种一致性的原因[33]：

> 我怀疑为什么我们对 ext3 如此之不及的真正原因是日志通常在磁盘上是连续的，因此，当您写入日志时，极不可能写入 commit 块并且在 commit 块之前的块没有。...但是，最重要的原因是脏的块不会立即冲出磁盘！

Ts’o 假设专门指的是两个顺序：JM→JC 和 JC→M。在第一种情况下，Ts’o 指出，即使没有中间刷新，磁盘也可能在 JM 之后将 JC 提交到磁盘（请注意，由于布局和计划的原因，这没有事务校验和）；磁盘根本不可能对连续的两次写入进行重新排序。在第二种情况下，Ts’o 注意到 JC→M 由于时间的关系经常保持平缓状态；将 M 提交到以下磁盘的检查点流量：

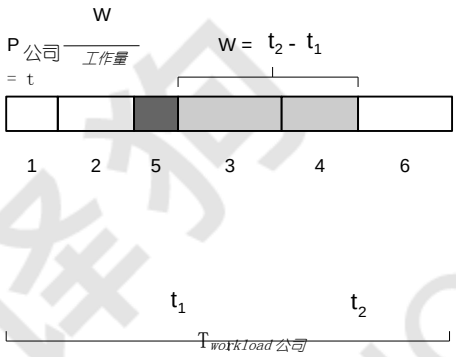

图 *2：不一致概率（Pinc）。显示了漏洞窗口的示例。块 1 到 6 严格按照顺序写入磁盘。但是，第 5 块（深灰色）是较早写入的。一旦提交了 5，则存在漏洞窗口，直到提交了块 3 和 4（浅灰色）为止；在此期间发生崩溃将导致可观察到的重新排序。不一致的概率由下式计算：*

用在这样的窗口中花费的时间（即 W = t2-t1）除以工作负载的总运行时间（即 tworkload）。

事务提交后很长时间发生了十次，因此不刷新就保留了顺序。

我们称这种安排为概率一致性。在这样的配置中，典型的操作可能会或可能不会导致太多的重新排序，因此磁盘有时仅处于不一致状态。尽管文件系统没有通过刷新命令强制执行崩溃，崩溃也不会导致不一致。尽管概率性崩溃一致性并不能保证崩溃后的一致性，但由于关闭冲洗功能会大大提高性能，因此许多从业者仍对它感兴趣。

3.1 量化概率一致性

不幸的是，概率一致性并没有被很好地理解。为了阐明这一问题，我们量化了不通过模拟冲洗而出现不一致的频率。为此，我们在 Linux ext4 中禁用刷新，并在一系列工作负载中提取 ext4 下的块级跟踪。我们会仔细分析这些痕迹，以确定由于崩溃而发生不一致的可能性。我们的分析是通过DiskSim [4]之上的模拟器完成的，该模拟器使我们能够对复杂的磁盘行为（例如，调度，缓存）进行建模。

我们仿真的主要输出是确定何时出现漏洞窗口（W），以及该窗口持续多长时间。由于重新排序而出现这样的窗口。例如，如果 A 应该在 B 之前写入磁盘，但是 B 在时间 t1 写入而 A 在 *t2* 时，系统状态容易受到 W = t2-t1 之间的时间段不一致的影响。如果系统在此窗口期间崩溃，则文件系统将保留
处于不一致状态；相反，一旦后者
（A）是书面的，不再有任何顾虑。

给定工作负载和磁盘模型，因此可以通过将在漏洞窗口中花费的总时间除以工作负载的总运行时间来量化不一致的可能性（Pinc）；图 2 显示了一个示例。请注意，当在启用了缓存刷新的文件系统上运行工作负载时，Pinc 始终为零。

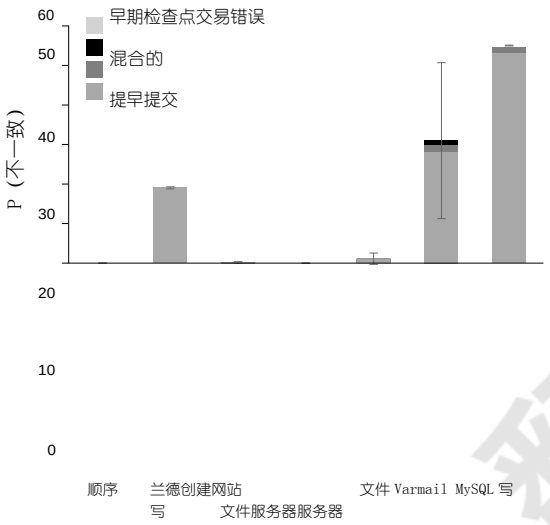

图 3：Pinc 的工作量研究。该图显示了用于六个工作负载的 Pinc。前两个工作负载是顺序写入和随机写入 1 GB 文件。Fileserver，Webserver 和 Var-mail 是 Filebench 基准套件的一部分[17]。Fileserver 执行一系列创建，删除，追加，读取和写入操作。Web 服务器模拟多线程 Web 主机服务器，对多个文件以及日志文件追加执行打开-关闭-关闭的顺序。Varmail 模拟多线程邮件服务器，在单个目录中执行一系列 create-append-sync，readappend-sync，reads 和 deletes 操作。MySQL 代表 Sysbench [1]提出的 OLTP 基准。每个条形图都细分为不同类型的错误排序的百分比贡献。还显示了标准偏差。

3.2 影响 P 公司的因素

现在，我们更加系统地探索 Pinc。具体来说，我们确定对工作负载和磁盘参数（例如队列大小和相对于文件系统结构的日志位置）的敏感性。我们使用 DiskSim 随附的经过验证的 Seagate Cheetah 15k.5 磁盘模型[4]进行实验。

3.2.1 工作量

我们首先展示工作负载如何影响 Pinc。对于本实验，我们使用图 3 标题中描述的 6 种不同的工作负载。从图中，我们得出以下观察结果。最重要的是，Pinc 与工作量有关。例如，如果工作负载主要是面向读取的，则由于文件系统状态不经常更新（例如，Web 服务器），因此几乎没有不一致的机会。第二，对于繁重的写工作负载，写操作的性质很重要。随机写入的工作负载或通过 fsync（）强制写入磁盘的工作负载导致崩溃的可能性很高，从而使文件系统不一致（例如，随机写入，MySQL，Varmail）。第三，Pinc 的差异很大。更改写入持久性顺序的小事件可能会导致不一致机会的差异很大。最后，即使在极端情况下，Pinc 也永远不会达到 100%（图表被截断为 60%）；当崩溃不会导致不一致时，在工作负载的生命周期中有很多要点。

除了图中显示的总体 Pinc 之外，我们还通过导致漏洞窗口的重新排序类型进一步打破了概率。具体来说，假设以下提交顺序（D | JM→JC→M），我们确定何时进行特定的重新排序（例如，JC

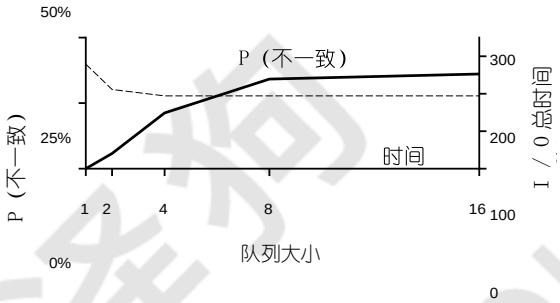

图 4：队列大小的影响。该图显示了 Pinc（左 y 轴）和总 I / O 完成时间（右 y 轴），因为模拟磁盘的队列大小有所变化（x 轴）。在本实验中，我们使用 Varmail 工作负载。

在 D）产生之前。该图将 Pinc 细分为这些细粒度的重新排序类别，分为以下相关情况：早期提交（例如，JC→JM | D），早期检查点（例如 M→D | JM | JC），transac-错误排序（例如 Txi→Txi-1）和混合错误（例如，可以归因于多个类别）。

我们的实验表明，先于数据提交（JC→D）是 Pinc 的最大贡献，占所有工作负载不一致的 90%以上，在某些情况下（文件服务器，随机写入）占 100%。这不足为奇，因为在将事务强制插入磁盘的情况下（例如，由于对 fsync（）的调用），数据写入（D）在交易写入之前发布（JM 和 JC）；磁盘的轻微重新排序将导致 JC 首先保留。另外，对于某些工作负载（MySQL，Varmail），所有类别都可能会造成影响。尽管很少见，但可能会出现早期的检查点和交易错误。因此，提供可靠一致性机制的任何方法都必须考虑所有可能的原因，而不仅仅是原因。

3.2.2 队列大小

对于其余的研究，我们将重点放在 Varmail 上，因为它展示了最有趣且变化最大的不一致概率。首先，我们展示磁盘调度程序队列深度的重要性。图 4 绘制了我们的实验结果。当我们更改对磁盘的未完成请求数时，左侧的 y 轴将绘制 Pinc。正确的 y 轴表示性能（所有 I / O 完成的总时间）。

从图中，我们观察到以下三个结果。首先，当磁盘没有完成重新排序（即队列大小为 1）时，就不会出现不一致的机会，因为写入是按顺序提交的；如果我们使用 FIFO 磁盘调度（而不是 SPTF），则会发现相同的结果。其次，即使队列很短（例如 8 个），也会出现大量不一致的情况。过早地将一个块提交到磁盘可能会导致很大的漏洞窗口。最后，我们观察到适量的重新排序确实确实会产生明显的性能差异。队列大小为 8 或更大时，磁盘内 SPTF 调度可将性能提高约 30%。

3.2.3 期刊布局

现在，我们研究主要文件系统结构和日志之间的距离如何影响 Pinc。图 5

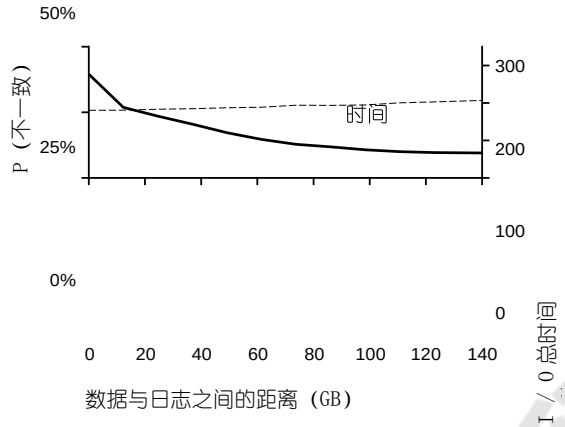

图 5：距离的影响。该图显示了 Pinc（左 y 轴）和总 I / O 完成时间（右 y 轴），随着数据区域和模拟磁盘的轴颈之间的距离（以 GB 为单位）增加（x 轴）。在本实验中，我们使用 Varmail 工作负载，队列大小设置为 8。

将 Varmail 的数据和元数据结构（通常位于一个磁盘区域中）的位置从靠近日志（左）到远处的不同位置绘制结果。

从图中可以看出，距离对 Pinc 的影响很大。回想一下重新排序的主要原因之一是提早提交（即，在 D 之前写的 JC）；通过将数据和日志的位置分开，这种重新排序的可能性越来越小。其次，我们还观察到距离的增加不是万能的。Varmail 仍然会出现不一致（10%）。最后，与轴颈的距离增加会在一定程度上影响性能。将 Varmail 的数据和元数据从日志旁边移到 140 GB 时，性能会降低 14%。

我们还研究了许多可能影响 Pinc 的其他因素，包括与磁盘上磁道边界相关的日志位置以及其他潜在因素。通常，这些参数不会显着影响 Pinc，因此不包括在内。

4.1 摘要

传统的日记记录方法过于悲观，通常仅在需要排序时才强制将其写入持久性存储。结果，用户有时会转向概率日记，他们会不断尝试一致性以获取更高的性能。我们已经仔细研究了哪些因素会影响概率方法的一致性，并表明对于某些工作负载，该方法效果很好。不幸的是，对于具有大量随机写入 I / O 的其他工作负载，或者在应用程序本身将流量压入磁盘的情况下，不一致的可能性会很高。随着设备变得越来越复杂，并且可以处理大量未解决的请求，崩溃将导致不一致的可能性增加。因此，为了超越概率方法，系统必须包括机械设备，以免导致导致不一致的情况，或者能够在这种情况发生时进行检测和恢复。现在，我们描述一种这样的方法：乐观的崩溃一致性。

4 乐观的崩溃一致性

鉴于具有概率一致性的日志记录即使在系统崩溃的情况下通常也能获得一致的结果，因此我们注意到了一个新的机会。在乐观日志系统中实现的乐观崩溃一致性的目标是将事务提交给持久化-

帐篷存储的方式可以保持与悲观日记相同的一致性，但性能几乎与概率一致性相同。乐观日志记录只需对当前磁盘接口和日志记录层进行最少的更改；特别是，我们的方法不需要更改日志外部的文件系统结构（例如，后向指针[6]）。

为了描述乐观的碰撞一致性和日志记录，我们首先描述乐观技术背后的直觉。乐观的崩溃一致性基于两个主要思想。首先，校验和可以消除对写入命令的需求。乐观的崩溃一致性通过将元数据事务校验和[23]概括为包括数据块，从而消除了在事务提交期间进行排序的需要。在恢复期间，如果校验和不匹配，事务将被丢弃。

其次，异步持久性通知用于延迟对事务的检查点，直到它被持久地提交为止。幸运的是，此延迟不会影响应用程序性能，因为应用程序会在事务提交之前（而不是在检查点之前）阻塞。在诸如块重用和覆盖之类的方案中，还需要其他技术来确保正确性。

我们首先建议磁盘驱动器应公开的其他通知。然后，我们解释乐观日记如何提供不同的属性以保留有序日记的一致性语义。我们表明，可以结合使用乐观技术来实现这些属性。我们还描述了一种额外的乐观技术，该技术可使乐观日记功能提供与数据日记功能等效的一致性。

4.1 异步耐久性通知

用于确保按指定顺序执行写操作的磁盘的当前接口是悲观的：上层文件系统告诉下层磁盘何时必须刷新其缓存（或某些块），然后该磁盘必须及时这样做。但是，写入盘的实际顺序和持久性并不重要，除非发生崩溃。因此，当前接口过于受限，并限制了 I / O 性能。我们建议不让磁盘服从上一层的命令和持久性命令，而是建议释放磁盘以按顺序执行读写，以优化其调度和性能。因此，磁盘的性能针对没有崩溃的常见情况进行了优化。

鉴于文件系统在发生崩溃时仍必须能够保证一致性和持久性，我们建议对磁盘接口进行最小扩展。用

磁盘通知的异步持久性通知

特定写请求已完成的高层客户端，现在可以保证持久。从而

磁盘上将有两个通知：首先是

磁盘已接收到写入，并且稍后写入已持久。某些接口（例如，强制单元访问（FUA））提供单个同步的持久性通知：驱动器接收到请求，并在请求被持久保存时指示完成[14，37]。标记排队允许在给定的时间点处理数量有限的请求[15，37]。不幸的是，许多驱动器不能可靠地实现标记排队和 FUA [18]。此外，标有 FUA 的请求也意味着紧迫性，促使某些实现将其立即强制到磁盘。尽管标记队列和 FUA 的正确实现可能足以实现乐观的崩溃一致性，但我们认为将请求确认与持久性脱钩的接口可以实现更高级别的 I / O 并发性，从而为构建 OptFS 提供了更好的基础。

4.2 乐观一致性属性

如第 2.2 节所述，有序日记记录模式涉及每个事务的以下写入：数据块 D，到磁盘上的就位位置；日志 JM 的元数据块；JC 的提交块；最后，是元数据到其就地位置的检查点 M。我们用符号 i 来指代属于特定事务 i 的写入。例如，事务 i 的日记化元数据表示为 JM：i。

有序日记记录模式可确保多个属性。首先，除非观察到来自事务 Tx：i 的元数据，否则无法观察到写入事务 Tx：i + 1 的元数据。其次，元数据不可能指向无效数据。这些属性由
恢复过程以及写入顺序。如果在正确提交事务之后（即，D，JM 和 JC 都被持久地写入），但是在写入 M 之前发生崩溃，则恢复过程可以重播事务，以便将 M 写入其就地位置。如果在事务完成之前发生崩溃，则有序日记记录可确保没有更新与该事务相关的就地元数据。

乐观日志允许磁盘以其选择的任何顺序执行写操作，但可以确保在发生崩溃的情况下，为有序事务保留必要的一致性属性。为了使读者有一个直观的认识，为什么特定属性足以满足有序的日记语义，我们遍历图 6 中的示例。为简单起见，我们首先假设数据

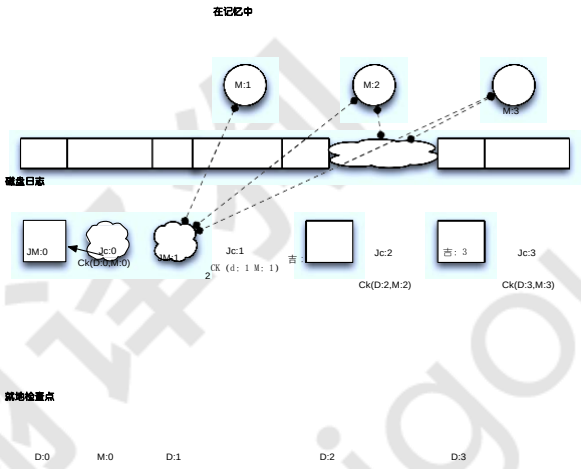

图 *6：乐观日记。该图显示了四个正在进行的事务，其中包括对主内存，磁盘日志以及磁盘上就地检查点的写入。矩形块指示已通知文件系统写入已持久完成。云形块表示已开始写操作，但尚未通知文件系统已完成写操作，它可能是持久的，也可能不是持久的。圆圈表示主存储器中的脏块，除非先前的写入操作是持久的，否则它们无法写入；虚线表示它所依赖的写入。最后，实线箭头表示元数据可以引用数据块。*

块不能跨事务重用（即，它们不会被释放并重新分配到不同的文件或被覆盖）；我们稍后将删除此假设（第 4.3.5 节）。

在图 6 中，正在进行四个事务：Tx：0，Tx：1，Tx：2 和 Tx：3。这时，文件系统已收到有关 Tx：0 持久的通知（即，D ：0，JM：0 和 JC：0），因此它正在将元数据 M：0 指向其在磁盘上的就位位置的过程（请注意，M：0 可能指向数据 D：0） 。如果此时发生崩溃，则恢复机制将正确地重播 Tx：0 并重新启动 M：0 的检查点。由于 Tx：0 是持久的，因此可以向发起这些写操作的应用程序通知该写操作已经完成（例如，如果它调用了 fsync（））。请注意，一旦通知文件系统 M：0 的就地检查点写入是持久的，就可以最终释放 Tx：0 的日志条目。

文件系统也已经开始事务 Tx：1 到 Tx：3;已经启动了许多相应的磁盘写操作，而其他一些则根据未解决的依赖关系保存在内存中。具体来说，已经开始了对 D：1，JM：1 和 JC：1 的写入。但是，D：1 尚未持久，因此可能无法引用指向它的元数据（M：1）。如果此时对 M：1 进行检查，并且发生了崩溃（M：1 被保留而 D：1 没有被保留），则 M：1 可能会指向 D：1 的垃圾值。如果现在崩溃，则在 D 之前：1 是持久的，添加到 Tx：1 的提交块中的校验和将表明与 D：1 不匹配；恢复过程将不会根据需要重播 Tx：1。

*T*x：2 允许与 Tx：1 并行进行；在这种情况下，尚未通知文件系统日志提交块 JC：2 已经完成；再次，由于事务尚不持久，因此无法检查元数据 M：2。如果在 JC：2 尚未持久时发生崩溃，则恢复过程将检测到数据块与校验和之间的不匹配，并且不会重新

播放 Tx: 2。请注意，D: 2 在这一点上可能是持久的，不会带来负面影响，因为还不允许元数据引用它，因此无法访问它。

最后，Tx: 3 也正在进行中。即使通知文件系统 D: 3，JM: 3 和 JC: 3 都是持久性的，也无法启动 M: 3 的检查点，因为 Tx: 1 和 Tx: 2 中的基本写入不持久（即，D: 1，JC: 2）。Tx: 只有保证所有先前的交易都是持久性的，才能使持久性 3；因此，其元数据 M: 3 无法检查点。

4.3 乐观技术

可以使用一组乐观技术来确保上述乐观日志的行为：有序日志恢复和释放，校验和，通知后的后台写，通知后重用以及选择性数据日志。现在我们描述每个。

4.3.1 有序日记帐恢复

在日志恢复过程本身发生崩溃后，用于保留正确的写入顺序的最基本技术。恢复过程读取日志以观察哪些事务变得持久，并且它简单地丢弃或忽略任何发生在所需顺序之外的写操作。

进行乐观日志记录的更正是确保如果事务 Tx: i 的任何部分未正确或完全不具有持久性，则事务 Tx: i 和随后的任何事务 Tx: j 都不具有持久性。因此，日记帐恢复必须按顺序进行，依次扫描日记帐并执行
按顺序检查点，在磁盘上未完成的第一个事务处停止。有序恢复过程将使用以下所述的校验和来确定事务是否正确且完整地写入。

4.3.2 有序日记帐发布

鉴于已完成的持久日志事务定义了在磁盘上持久的写操作，因此乐观日志记录必须确保在确认所有相应的（元数据）检查点写入为持久之前，日志事务不会被释放（或覆盖）。

因此，乐观日志必须等待，直到磁盘已通知它与该事务对应的检查点写入是持久的为止。在这一点上，乐观日志知道如果系统崩溃，则无需重播事务。因此，可以释放交易。要保存的财产
*T*x: i + 1 仅在 Tx: i 是持久性的情况下才是持久性的，必须按顺序释放事务。

4.3.3 校验和

校验和是一种众所周知的技术，用于检测数据损坏和丢失的写入[20，31]。校验和还可以用于检测写入是否与
发生了特定的交易。具体来说，校验和可以乐观地“强制执行”两个顺序：日志提交块（JC）仅在元数据到达日志（JM）之后以及数据块到达其就地位置（D）之后才继续存在。这种确保元数据被持久地完整写入日志的技术被称为事务校验和[23]。在这种方法中，将通过 JM 计算校验和并将其放在 JC 中。如果在提交过程中发生崩溃，则恢复过程可以可靠地检测到 JM 与 JC 中的校验和之间的不匹配，而不会重播该事务（或随后的任何事务）。为了标识此事务校验和的特定实例，我们将其称为元数据事务校验和。

可以使用类似但更复杂的数据事务校验和版本来确保将数据块 D 作为事务的一部分完整写入。与在 JM 上执行校验和相比，收集这些数据块并计算它们在主内存中被弄脏时的校验和要比在 JM 上执行校验和更为复杂，但是基本思想是相同的。通过将数据校验和及其磁盘上的块地址存储在 JC 中，日志恢复过程可以在不匹配时中止事务。因此，数据事务校验和使乐观日记功能可以确保在未持久写入相应数据的情况下不会对元数据进行检查。

4.3.4 通知后后台写

一种重要的乐观技术可确保元数据（M）的检查点在先前写入数据和日志（即 D，JM 和 JC）之后发生。悲观日记在 JC 之后通过刷新来保证这种行为，而乐观日记显式推迟了元数据 M 的检查点写入，直到已通知所有先前的事务已持久完成。注意仅在 JC 之后发生 M 是不够的;D 和 JM 也必须先于 M，因为乐观日记必须确保整个交易有效，并且如果有任何以下情况，则可以重播：
写入位置元数据 M。同样，必须将 M: i + 1 推迟到所有事务 Tx: i 为止，如果 M: i 无法重播，则必须确保 M: i + 1 不持久。我们注意到，M: i +1 无需等待 M: i 完成，而必须等待
负责任的交易 Tx: 我要持久。检查点是优化中的少
数几个点之一
文件系统必须在其中的 mistic 日志记录协议
等待发出特定的写入，直到完成特定的写入集为止。但是，这种特殊的等待不太可能影响性能，因为检查点发生在后台。应用程序随后的写入将被放置在以后的事务中，并且这些日记更新可以独立于任何其他未完成的写入进行写入；日记写不需要

等待先前的检查点或事务。因此，即使等待日志写入的应用程序（例如，通过调用 fsync（））也不会观察到检查点延迟。

因此，在后台检查点之前等待异步持久性通知从根本上比向磁盘发送排序命令（即，缓存刷新）的悲观方法更强大。使用传统的排序命令，磁盘将无法推迟跨多个事务的检查点写入。另一方面，异步持久性通知命令不会人为地限制写入的顺序，并为磁盘提供更大的灵活性，以便它可以最佳地缓存和调度写入。该命令还向文件系统提供所需的信息，以便它可以允许独立写操作继续进行，同时适当地阻止依赖写操作。

4.3.5 通知后重用

前面的技术足以处理跨事务未重用块的情况。困难在于有序日志记录，因为数据写入到它们的就地位置，有可能覆盖磁盘上仍被先前事务中的持久元数据引用的数据。因此，需要其他乐观技术来确保来自较早事务的持久元数据永远不会指向在较晚事务中更改的不正确数据块。这是一个安全问题：如果用户 A 删除了他们的文件，然后删除的块成为用户 B 的文件的一部分，则崩溃不应导致用户 A 能够查看用户 B 的数据。这也是软更新[25]所需的更新依赖规则之一，但是乐观日志通过另一种技术强制执行此规则：通知后重用。

为了理解该技术的动机，请考虑将数据块 DA 从一个文件 MA 中释放并分配给另一个文件 MB 并用内容 DB 重写时的步骤。根据写入对磁盘的重新排序方式，MA 的持久版本可能指向 DB 的错误内容。

可以通过以下方法解决此问题。首先，释放 DA 并更新为 MA（表示为 M_A'），
作为事务 JM_A，的一部分编写：i;DB 到 MB 的分配将在以后的事务中写为 DB：i + 1 和 JM_B：i +1。悲观日志记录确保 JM_A，：i 出现在 DB：i + 1 之前，并且在
每笔交易。乐观技术介绍
到目前为止，这还不足以提供此保证，因为如果丢失了 M′（即使校验和不匹配且事务 Tx：i），则无法恢复或回滚就地写入 DB：i_A+ 1 的数据。或 Tx：发现 i + 1 不完整）。

乐观日志记录通过确保数据块 DA 不重新出现来确保 JM_A，：i 出现在 DB：i + 1 之前分配给另一个文件，直到磁盘通知文件系统 JM_A，：i 已被持久写入；此时，数据块 DA 是“持久的”。当必须为文件 MB 分配一个新的数据块时，乐观文件系统将分配一个“持久的”数据块，已知该数据块不会被任何其他文件引用;鉴于拟议的磁盘异步持久性通知，找到一个持久可用的数据块很简单。

仅在通知之后执行重用不太可能导致文件系统等待或损害性能。除非文件系统将近 100%已满，否则应始终存在一个已知持久可用的数据块列表。在正常操作条件下，文件系统不太可能需要等待磁盘通知特定数据块可用。

4.3.6 选择性数据日记

我们最终的乐观技术有选择地记录已被覆盖的数据块。此技术允许乐观日记提供数据日记一致性语义，而不是有序日记语义。

当文件中的数据块被覆盖并且该文件的元数据（例如，大小）必须一致更新时，就会发生更新依赖的特殊情况。乐观日记可以在通知后使用重用来处理此问题：将新的持久数据块分配给文件并写入磁盘（DB：j），然后记录新的元数据（JM_B：j）。这种方法的缺点是文件系统承担了写时复制文件系统的行为，并失去了就地更新文件系统的一些本地性优势[21]；由于乐观日志记录会强制分配一个持久无用的数据块，因此对于某些随机更新工作负载而言，最初连续分配并提供高顺序读取和写入吞吐量的文件可能会失去其位置。

如果需要就地更新以提高性能，则可以使用其他技术：选择性数据日志记录。数据日记将元数据和数据都放置在日记中，然后在检查点时间就地进行更新。数据日记功能的吸引人之处在于，在对事务进行检查之前，就不会覆盖就地数据块。因此，如果数据块的元数据也在同一事务中更新，则可以重用这些数据块。数据日志的缺点是每个数据块都被写入两次（一次在日志 JD 中，一次在其检查点就位 D），因此其性能通常比有序日志差[22]。选择性数据日志记录允许将有序日志记录用于常见情况，仅当数据块在同一文件中反复被覆盖且文件需要保持其在磁盘上的原始布局时，才使用数据日志记录。

在选择性数据日记中，D 和 M 的检查点都只是在等待所有对象的持久通知该期刊写道（JD，JM 和 JC）。

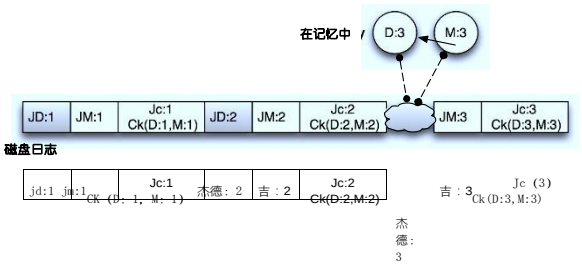

图 7：乐观日记：选择性数据日记。该图显示，当事务涉及覆盖就地数据时，可以使用选择性数据日志记录。现在，将数据块放置在日志中，并在提交事务后检查点。

图 7 显示了一个示例，该示例说明了如何使用选择性数据日志记录来支持覆盖，特别是在从先前的事务中重用块而不清除对这些块的原始引用的情况下。在此示例中，三个文件的数据块已在三个单独的事务中被覆盖。

第一个事务说明了乐观排序如何确保持久性元数据不指向垃圾数据。在通知文件系统 Tx：1（特别是 JD：1，JM：1 和 JC：1）的持久性之后，它可以将 D：1 和 M：1 都指向其就位位置。因为文件系统可以写 M：1 而无需等待 D：1 的持久通知，所以在崩溃的情况下，M：1 可以引用 D：1 中的垃圾值；但是，恢复过程将由于校验和不匹配而识别这种情况，并使用正确的 D：1 重放 Tx：1。

第二和第三次事务说明乐观排序如何确保仅当所有较早的写入也可见时才可看见较晚的写入。具体来说，D：2 和 M：2 已经过检查点，但这仅仅是因为 Tx：2 和 Tx：1 都是持久的；因此，客户端无法看到第二个文件的新内容，而看不到第一个文件的新内容。而且，D：3 或 M：3（或任何以后的事务）都不能被检查点，因为并不是所有的事务块都是持久的。因此，选择性数据日志提供了所需的一致性语义，同时允许覆盖。

4.4 耐用性与一致性

乐观日志记录使用了一系列新颖的技术来确保对磁盘的写操作正确排序，或者确保磁盘上存在足够的信息，以便在不按顺序发出写操作时从不合时宜的崩溃中恢复过来；结果是文件系统的一致性和正确的写入顺序，但不能保证持久性。但是，某些应用程序可能出于耐用性而不是有序的目的而希望强制写入稳定的存储区。在这种情况下，除了乐观排序之外，还需要其他一些东西。文件系统必须等待此类写操作得以持久（通过异步持久性通知），或者是-

起诉冲洗，以强制问题。为了区分这些情况，我们认为应该提供两个电话；“排序”同步 osync（）确保写入之间的顺序，而“耐久性”同步 dsync（）确保何时返回未处理的写入。

现在，我们定义并比较 osync（）和 dsync（）给出的保证。假设用户进行了一系列写操作 W1，W2，...，Wn。如果没有 osync（）或 dsync（）

进行调用，无法保证文件系统
崩溃后的状态：任何或所有更新可能会丢失，并且更新可能会无序应用，即 W2 可能没有 W1 而被应用。

现在考虑何时每次写入后都跟随 dsync（），即 W1，d1，W2，d2，...，Wn，dn。如果在 di 之后发生崩溃，文件系统将恢复到应用了 W1，W2，...，Wi 的状态。

如果每个写操作后都跟随 osync（），即 W1，o1，W2，o2，...，Wn，然后在 oi 之后发生崩溃，则文件系统将恢复到 W1，W2 ..的状态. .，Wi-k 适用，最后 k 次未写入
崩溃前坚固耐用。我们将这种最终耐久性称为“持久性”。因此，osync（）提供了前缀语义[36]，即使数据可能是陈旧的，也可以确保用户始终看到文件系统的一致版本。先前的研究表明，这在许多领域都是有用的[7]。

5 OptFS 的实施

根据 ext4 文件系统的变体，我们已在 Linux 3.2 的基础上根据§4 概述的原理实现了乐观文件系统（OptFS），并对 JBD2 日记记录层和虚拟内存子系统进行了其他更改。

5.1 异步耐久性通知

由于当前磁盘未实现建议的异步耐用性通知接口，因此 OptFS 使用以下近似值：耐用性超时。耐久性超时表示磁盘可以延迟将写入请求提交到非易失性磁盘的最大时间间隔。当在时间 T 接收到对块 A 的写操作时
磁盘，必须在时间 T +之前使该块持久
ZD。TD 的值特定于每个磁盘，并且必须由磁盘导出到存储堆栈中的更高级别。

在时间间隔 TD 到期后，OptFS 认为该块是持久的；这等效于磁盘在 TD 秒后通知 OptFS。请注意，要考虑到 I / O 子系统中的其他延迟，TD 是从磁盘确认写入时开始测量的，而不是从文件系统发出写入时开始的测量。

这种近似是限制性的。TD 可能会高估耐用性间隔，从而导致性能问题和内存压力。TD 可能会低估耐久性，包括一致性。OptFS 会提高安全性，并将 TD 设置为 30 秒。

5.2 处理数据块

OptFS 不会记录所有数据块：仅对新分配的数据块进行校验和；它们的内容不存储在日志中。记录校验和后，数据块的内容可能会更改，这会使日志恢复变得更加复杂。由于有选择地进行数据日志记录，在一个事务中未记录日志（因为它是新分配的）的数据块可能在下一个事务中被覆盖并因此被记录。例如，考虑具有内容 A 属于 Tx1 的数据块 D。校验和 A 将记录在 Tx1 中。D 被 Tx2 和内容 B 覆盖。尽管此序列有效，但 Tx1 中的校验和 A 与 D 中的内容 B 不匹配。

这需要单独的块校验和，因为
如果该块属于以后的有效事务，则单个块的校验和不匹配不是问题。相反，由于元数据块的冻结状态存储在日志中，因此整个集合的校验和对于元数据事务校验和就足够了。我们将在短期内说明 OptFS 如何处理此问题。

OptFS 不会立即写出校验和的数据块。它们被收集在内存中，并在事务提交时成批写入。这样可以提高某些工作负载的性能。

5.3 乐观技术

现在，我们描述乐观日记技术的实现。我们还描述了 OptFS 在某些情况下（例如，当日志空间不足时）如何恢复为更传统的机制。

有序日记帐恢复：OptFS 按照提交到日记帐的顺序恢复事务。仅当事务的所有数据块都属于有效事务，并且通过元数据块计算出的校验和与提交块中的校验和匹配时，才可以重播该事务。

OptFS 分两步执行恢复：第一遍线性扫描日志，编译校验和不匹配的数据块列表，以及包含每个块的第一日志事务。如果以后的有效事务与块校验和匹配，则将该块从列表中删除。扫描结束时，将记录列表中最早的事务。下一遍将执行日志恢复，直到出现故障事务，从而确保一致性。

OptFS 日志恢复可能比 ext4 恢复要花费更长的时间，因为 OptFS 可能需要从非连续磁盘位置读取数据块，而 ext4 只需要读取连续日志以执行恢复。

有序日记帐发布：当虚拟内存（VM）子系统通知 OptFS 在时刻 T 已确认检查点块时，OptFS 将事务清除时间设置为 T + TD，之后将其从日记中释放。

当日志空间不足时，等待耐久性超时间隔的时间可能不是最佳的-
释放日志块。在内存压力下，OptFS 可能需要释放已发布到磁盘并等待持久性超时的检查点块的内存缓冲区。在这种情况下，OptFS 发出磁盘刷新操作，以确保磁盘已确认的检查点块的持久性。这使 OptFS 可以清除属于某些检查点事务的日志块并释放关联的内存缓冲区。

校验和：OptFS 校验和数据块使用与元数据相同的 CRC32 算法。为每个数据块创建一个标签，该标签存储块号及其校验和。数据标签与元数据校验和的标签一起存储在描述符块中。

通知后的后台写入：OptFS 使用 VM 子系统执行后台写入。检查点元数据块被标记为脏，每个块的到期字段设置为 T + TD（磁盘在 T 处确认提交块）。T 将反映
这是因为在磁盘确认数据和元数据写入之前才发出提交，因此已确认整个事务。然后将这些块移交给 VM 子系统。

在定期进行后台写操作期间，VM 子系统会检查每个脏块是否已到期：否则，VM 子系统会在下一个定期写出周期中重新检查该块。

通知后重用：提交事务后，OptFS 将已删除的数据块添加到全局内存块列表中，该列表将在持久性超时 TD 之后释放。后台线程会定期释放具有过期的超时超时的块。卸载文件系统后，将释放所有列表块。

当文件系统空间不足时，如果重用列表包含块，则发出磁盘刷新；这样可以确保释放列表中的块的事务的持久性。然后将这些块设置为空闲。我们希望这些“安全”冲洗将很少使用。

选择性数据日志记录：在写入块时，块分配信息（反映在缓冲区头的“ New”状态中）用于确定是否重新分配了该块。-如果写操作是覆盖操作，则记录该块，就像它是元数据一样。

6 评价

现在，我们在两个方面评估 OptFS 的原型实现：可靠性和性能。实验是在具有 16 GB 内存的 Intel Core i3-2100 CPU，Hitachi DeskStar 7K1000.B 1 TB 驱动器以及运行 Linux 3.2 的环境下进行的。实验是可重复的。报告的数字是 10 次运行的平均值。

工作量	延迟块	碰撞点	
		总	一致的
附加	数据	50	50
	m	50	50
	多个块	100	100
覆写	数据	50	50
	m	50	50
	多个块	100	100

表 1：可靠性评估。该表显示了模拟碰撞点的总数，以及重新安装后导致状态保持一致的碰撞点数。

6.1 可靠性

为了验证 OptFS 的一致性保证，我们在两个综合工作负载下构建并应用了一个崩溃鲁棒性框架。第一个将块附加到文件末尾；第二个将块附加到文件末尾。第二个覆盖现有文件的块；两者都频繁发出 osync（）调用，以引发更多的订购点，从而给 OptFS 机械施加压力。

通过进行工作负载跟踪，重新排序一些请求（通过延迟单个数据块，日志提交或元数据块的写入或随机选择的多个块的写入），创建包含以下内容的文件系统映像来执行崩溃模拟：这些文件会一直写到特定的崩溃点，然后再从该映像恢复文件系统。

表 1 显示了 400 种不同崩溃情况的结果。OptFS 在每种情况下都可以正确地恢复到具有一致更新前缀（第 4.4 节）的文件系统。

6.2 性能

现在，我们在许多微观和宏观基准下分析 OptFS 的性能。图 8 说明了这些工作负载下的 OptFS 性能。在标题中可以找到工作负载的详细信息。

微观基准：OptFS 顺序写入性能类似于有序模式；但是，顺序覆盖会导致带宽下降到有序模式带宽的一半，因为 OptFS 将每个块写入两次。OptFS 上的随机写入速度比 ext4 有序模式下的写入速度快 3 倍，因为 OptFS 将随机覆盖转换为顺序日志写入（由于选择性数据日志记录）。

在 Createfiles 基准测试中，OptFS 的性能比 ext4 有序模式高 2 倍，这是因为 ext4 在后台将脏块写入后台有多种原因（例如，达到内存中脏数据量的阈值），而 OptFS 则将写入和合并整合在一起。在提交时发布它们。当我们修改 ext4 以停止后台写入时，其性能类似于 OptFS。

宏基准测试：我们运行 Filebench Fileserver，Webserver 和 Varmail 工作负载[17]。OptFS 的执行与 ext4 有序模式相似，但不刷新 Fileserver 和 Webserver。Varmail 频繁的 fsync（）调用会导致大量刷新，从而使 OptFS 的性能比 ext4 好 7 倍。ext4,

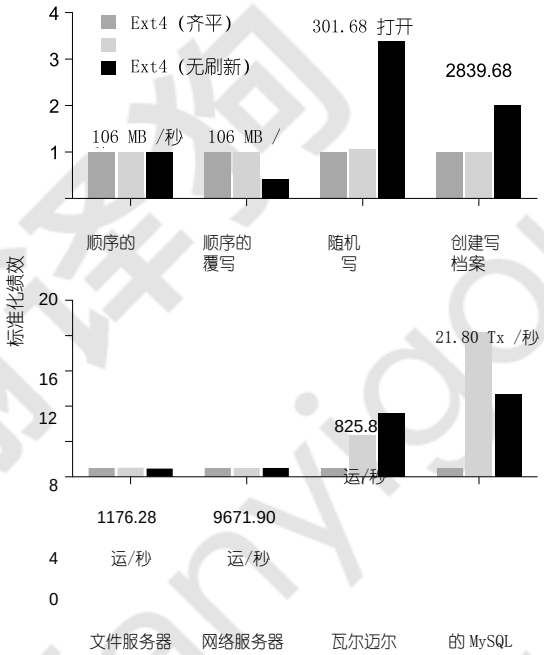

图 8：性能比较。显示性能已标准化为带有刷新的 ext4 有序模式。带有刷新的有序模式的绝对性能显示在每个工作负载上方。顺序写入 80 GB 文件。在 10 GB 的文件上执行 200K 随机写入，每写入 1K 写入一次 fsync（）。覆盖基准测试将顺序写入 32 GB 的文件。Createfiles 使用 64 个线程来创建 1M 文件。Fileserver 模拟文件服务器活动，使用 50 个线程执行一系列创建，删除，附加，读取和写入操作。Web 服务器模拟多线程 Web 主机服务器，对具有 100 个线程的多个文件加上日志文件追加执行打开-关闭-关闭的顺序。Varmail 模拟多线程邮件服务器，在单个目录中执行一系列 create-appendsync，read-append-sync，读取和删除操作。每个工作负载运行 660 秒。MySQL OLTP 基准测试对具有 1M 行的表执行 200K 查询。

即使禁用了刷新功能，由于 OptFS 会延迟写入脏块，定期或在提交时大批量发布脏块，因此其性能也不会达到 OptFS。相反，ext4 问题中的后台线程以小批量写入，因此不会影响前台活动。

最后，我们运行 Sysbench [1]的 MySQL OLTP 基准测试来研究数据库工作负载的性能。OptFS 的性能比带刷新的有序模式好 10 倍，比不带刷新的有序模式差 40%（由于 MySQL 的许多就地覆盖，导致选择性数据记录）。

简介：OptFS 在大多数工作负载上均具有刷新功能，明显优于有序模式，以较低的成本提供了相同级别的一致性。在许多工作负载上，OptFS 的性能与无刷新的有序模式一样好，因此无法提供一致性保证。OptFS 可能不适用于主要由顺序覆盖组成的工作负载。

6.3 资源消耗

表 2 比较了 660 秒运行的 Varmail 的 OptFS 和 ext4 的资源消耗。与没有刷新的 ext4 有序模式相比，OptFS 消耗的 CPU 多 10%。我们原型中的一些开销可以是

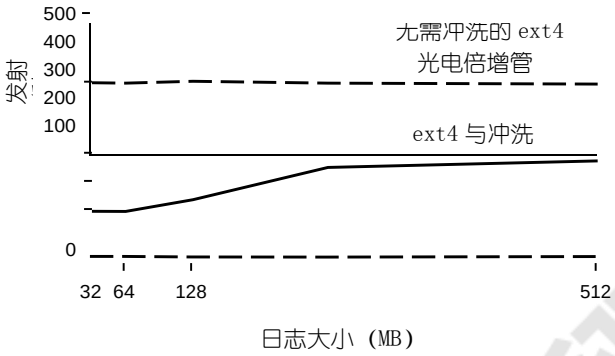

图 9：小型日记本的性能。无花果

图中显示了随着日志大小的变化，MySQL OLTP 基准上 OptFS 性能的变化。当 OptFS 用完日志空间时，它将发出刷新以安全地检查点事务。请注意，即使在这种压力情况下，OptFS 性能也比使用刷新的 ext4 有序模式高 5 倍。

归因于 CRC32 数据校验和和后台线程，该线程释放具有过期持久性超时的数据块，但需要进一步研究。

OptFS 延迟检查点并将元数据保存在内存中的时间更长，从而增加了内存消耗。而且，OptFS 将数据写入延迟到事务提交时间之前，从而提高了某些工作负载（例如Filebench Createfiles）的性能，但以增加内存负载为代价。

6.4 期刊尺寸

当 OptFS 用完日志空间时，它将发出刷新以检查点事务和释放日志块。为了调查这种情况下 OptFS 的性能，我们减小了日志大小并运行 MySQL OLTP 基准测试。结果如图 9 所示。请注意，由于具有选择性日志记录，因此即使有大日志记录，OptFS 性能也将比不带刷新的 ext4 更低。我们发现 OptFS 在合理大小的 512 MB 及更大的日志上表现良好。只有使用较小的日志大小，性能才会随着刷新而降低到 ext4 的水平。

7 实例探究

现在，我们展示如何使用 OptFS 排序保证来提供有意义的应用程序级崩溃一致性。具体来说，我们在文本编辑器（gedit）中研究原子更新，并在数据库（SQLite）中记录日志。

7.1 Gedit 中的原子更新

许多应用程序按以下顺序自动更新文件：首先，以临时名称创建文件的新版本；其次，在文件上调用 fsync（）以将其强制到磁盘；第三，将文件重命名为所需的文件名，用新的内容原子替换原始文件（某些应用程序将另一个 fsync（）发出到父目录，以保留名称更改）。我们将在此处研究的 gedit 文本编辑器执行此顺序以更新文件，以确保旧内容或

文件系统	中央处理	记忆体（MB）
带有冲洗的 ext4 有序模式	3.39	486.75
ext4 有序模式，不刷新	14.86	516.03
光电倍增管	25.32	749.4

表 2：资源消耗。下表显示了

对于 660 秒运行的 Filebench Varmail，OptFS 和 ext4 有序模式会浪费大量资源。由于延迟检查点之类的技术，OptFS 产生了额外的开销。

新内容都完整存在，但绝不混合。

	格迪特			SQLite 的		
	没有 ext4	Ext4 带/	光电倍增管	没有 ext4	Ext4 带/	光电倍增管
总崩溃点	50	50	50	100	100	100
前后不一致	7	0	0	73	0	0
旧状态	26	21	36	8	50	76
新状态	17	29	14	19	50	24
每操作时间（毫	1.12	39.4	0.84	23.38	152	15.3

表 *3: 案例研究: Gedit 和 SQLite。该表显示了重新挂载后导致应用程序状态一致或不一致的模拟崩溃点的数量。它还显示了应用程序操作所需的时间。*

为了研究 OptFS 在这种情况下的有效性，我们将 gedit 修改为使用 osync () 而不是 fsync ()，从而确保保留顺序。然后，当在ext4（带和不带刷新）和 OptFS 下运行时，我们进行块级 I / O 跟踪，并模拟大量崩溃点和 I / O 重新排序，以便确定在一次故障期间发生了什么。崩溃。具体来说，我们创建一个磁盘映像，该映像对应于崩溃之前发生的特定写入子集。然后，我们挂载映像，运行恢复，并测试正确性。

表 3 显示了我们的结果。使用 OptFS，保存的文件名始终始终指向整个数据的旧版本或新版本。实现原子更新。在相当数量的崩溃中，由于 OptFS 延迟写入更新，因此会恢复旧内容。这是 OptFS 所做的基本折衷，可以提高性能，但会延迟耐用性。使用 ext4（不刷新）时，大量崩溃点会导致不一致，包括无法安装的文件系统和损坏的数据。正如预期的那样，ext4（带冲洗）效果更好，导致新内容或旧内容完全符合 fsync () 边界的要求。

表 3 的最后一行比较了 OptFS 和 ext4 中原子更新的性能。OptFS 的性能与不带 ext4 的 ext4 相似，每次操作的速度大约比不带 ext4 的 ext4 高 40 倍。

7.2 临时登录 SQLite

现在，我们研究在数据库管理系统 SQLite 中对 osync () 的使用。为了实现 ACID 事务，SQLite 首先创建一个临时日志文件，向其中写入一些信息，然后调用 fsync ()。然后，SQLite 会就地更新数据库文件，对数据库文件调用 fsync ()，最后删除日志文件。之后

崩溃，如果存在日志文件，SQLite 将使用该日志文件进行恢复（如有必要）；确保将数据库恢复到交易前或交易后状态。

尽管 SQLite 事务默认提供持久性，但其开发人员断言许多情况不需要它，并且在这种情况下可以用纯顺序点代替“同步”。以下是 SQLite 文档[29]的摘录：

> 只要在同步之前发生的所有写入都在同步之后发生的任何写入之前完成，就不会发生数据库损坏。该数据库至少将继续保持一致，这就是大多数人关心的（强调我们的）。

现在，我们对 SQLite 进行相同的一致性和性能测试。我们使用一小组表（大小约为 30KB）创建一个事务，以将记录从表的一半移到另一半。装入与特定崩溃点相对应的模拟磁盘映像后，如果一致，则 SQLite 应该转换
数据库处于事务处理前（旧）状态或事务处理后（新）状态。表 3 中的结果类似于 gedit 案例研究：OptFS 始终处于一致状态，而没有刷新的 ext4 则不会。进行刷新时，OptFS 的性能比 ext4 好 10 倍。

8 相关工作

建立高性能文件系统的许多方法与我们在 OptFS 和乐观崩溃一致性方面的工作有关。例如，“软更新[11]”显示了如何仔细排序磁盘更新，以便永远不会以不一致的形式保留磁盘结构。与 OptFS 相比，FreeBSD Soft Updates 发出刷新以实现 fsync（）和排序（尽管原始工作修改了 SCSI 驱动程序以避免发出刷新）。给定异步持久性通知的存在，可以对软更新进行修改以利用此类信号。我们认为，这样做比修改日记文件系统更具挑战性。日记功能在元数据和数据的抽象级别上运行，而软更新直接在文件系统结构上运行，从而大大增加了其复杂性。我们的工作与 Frost 等人在 Featherstitch [10]上的工作相似，后者以软更新或基于日志的方法提供了用于命令文件系统更新的通用框架。相反，我们的工作重点是延迟期刊提交的排序；我们的某些技术可以提高期刊的绩效在他们的工作中观察到。

Featherstitch 为排序和持久性提供了类似的原语：pgDepend（）与 osync（）类似，而 pg sync（）与 dsync（）类似。主要区别在于应用程序所需的工作量。应用程序开发人员：Featherstitch 要求开发人员将文件系统操作集显式封装到称为补丁组的单元中，并定义它们之间的依赖关系。由于 osync（）建立在 fsync（）熟悉的语义基础上，因此我们相信应用程序开发人员将更易于使用。

关于“重新同步”的其他工作[19]与我们的工作具有相似的风格。在这项工作中，作者巧妙地指出，只有当某些外部实体可以观察到磁盘的持久性时，磁盘写入才需要变得持久。因此，通过将持久性延迟到发生这种外部化之前，可以实现性能的巨大提升。我们的工作是互补的，因为它减少了此类持久事件的数量，而在写入之间强制执行了较弱（和更高的性能）排序，但避免了在 OS 中实现依赖项跟踪的复杂性。在需要持久性的情况下（即应用程序使用 dsync（）而不是 osync（）），乐观日志记录不会带来太多收益；因此，Nightingale 等人的工作仍然可以从中受益。

最近，Chidambaram 等人。实现 NoFS [6]，从而根本不需要对磁盘进行任何订购，从而提供了出色的性能。但是，缺少有序的写入意味着某些类型的崩溃可能导致恢复的文件系统一致，但其中包含部分完成的操作的数据。结果，NoFS 无法实现原子操作，例如 rename（）。在以下更新顺序中，原子重命名至关重要：创建临时文件；写入包含旧文件的全部内容以及更新的临时文件；通过 fsync（）持久保存临时文件；自动将临时文件重命名为旧文件。在此序列的最后，文件应以所有更新的旧状态或新状态存在。如果 rename（）不是原子的，或者可以对操作进行重新排序，则整个文件可能会由于不适当的崩溃而丢失（在部署中已经观察到[9]）。相比之下，OptFS 实现了 NoFS 方法的许多优点，但是仍然提供了有意义的语义，可以在此语义上构建更高级别的应用程序语义。

一些实际的系统向应用程序提供了不同的 fsync（）风格。例如，在 Mac OS X 上，fsync（）仅确保已从主内存向磁盘发出写操作。如手册页所述，如果应用程序需要持久性，则应使用 F FULLSYNC 标志调用 fcntl（）。尽管后者与 dsync（）相同，但前者不等同于 osync（）；重要的是，仅刷新对磁盘缓存的写操作并不能提供任何更高级别的顺序保证，因此不能用作构建应用程序一致性的合适原语；相反，如我们所示，osync（）非常适合这种使用情况。

分布式系统中的先前工作旨在衡量最终一致性（在副本之间）并了解

技术	一致性	性能可用性	可用性 TX	支持	事务性
文件系统检查元数据日志数据日志软更新	L	H	L	√	√
	M	M	H	√	×
	H	M	H	√	×
写时复制	M	M	H	√	✗
	H	M	H		✗
基于 Backpointer 的一致性乐观崩溃一致性	M	H	L	√	√
	H	H	H*	√	√

表 4：一致性技术。该表比较了提供文件系统一致性的各种方法。图例：L —低，M —中，H —高。H 表示，如果需要的话，乐观的崩溃一致性可以使用 dsync () 提供即时的持久性。请注意，只有乐观的崩溃一致性才能提供事务支持，按需即时持久性和高性能，同时在通常情况下消除了刷新。*

它在实践中为何起作用，类似于概率崩溃一致性试图了解文件系统如何在不刷新的情况下保持崩溃一致性的原因。Yu 等。提供度量标准来衡量副本之间的一致性，并表明这种量化允许复制的服务进行有益的权衡[39]。Bailis 等。在最终一致的分布式系统中限制了数据的陈旧性，并解释了为什么最终的一致性在实践中有效[2]。

最后，我们将乐观的崩溃一致性与其他在文件系统中提供崩溃一致性的方法进行了比较。我们已经详细讨论了其中的一些，例如软更新和基于 Backpointer 的一致性。表 4 在耐久性，可用性和性能等方面广泛地比较了各种一致性技术。事务性支持表示支持多块原子操作，例如 rename ()。无刷新表示在通常情况下该技术不会发出刷新。

我们观察到有四种乐观的方法：文件系统检查[5]，软更新，基于Backpointer的一致性和乐观崩溃一致性。**“软更新”**会发出刷新来确保订购，因此并非并非免费。尽管文件系统检查和基于后向指针的一致性是完全免刷新的，但它们不能强制写入磁盘，因此最终具有持久性。日志记录和写时复制之类的悲观方法采用冲洗来提供事务支持以及高水平的一致性和持久性。但是，如果没有刷新，他们将无法实现订购，因此，在具有许多订购点的工作负载上，性能会降低。由于使用了 osync () 和 dsync ()，乐观的崩溃一致性可以按需持久写入（导致即时的持久性），但是在仅需要订购时保持免刷新状态。

9 结论

我们提出了乐观的崩溃一致性，这是一种在日记文件系统中实现崩溃一致性的新方法，该方法使用了一系列新颖的技术来获得高水平的一致性和在常见情况下的出色性能。我们介绍了两种新的文件系统原语 osync () 和 dsync ()，它们将顺序与持久性脱钩，并显示它们可以为应用程序开发人员提供高性能和有意义的语义。我们认为，这种解耦是解决文件系统中一致性和性能之间持续不断的紧张关系的关键。

OptFS 的源代码可在以下位置获得：
`http://www.cs.wisc.edu/adsl/Software/optfs`.
我们希望这将鼓励采用乐观的方法来保持一致性。

致谢

我们感谢匿名评论者 James Mickens（我们的牧羊人）和 Eddie Kohler 的深刻见解。我们感谢 Mark Hill，Michael Swift，Ric Wheeler，Jooyoung Hwang，Sankaralingam Panneerselvam，Mohit Saxena，Asim Kadav，Arun Kumar 和 ADSL 实验室的成员的反馈。我们感谢 Sivasubramanian Ramasubramanian 帮助我们运行了概率碰撞一致性实验。该材料基于 NSF 在 CNS-1319405 和 CNS-1218405 下的支持以及 EMC，Facebook，Fusion-io，Google，华为，Microsoft，NetApp，Sony 和 VMware 的捐赠。本文所表达的任何意见，发现和结论或建议均为作者的观点，不一定反映了 NSF 或其他机构的观点。

参考文献

[1] Alexey Kopytov。SysBench：系统性能基准。http://sysbench.sourceforge.net/index. HTML格式,2004.

[2] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein和Ion Stoica。实用部分定额的概率有界的陈旧性。PVLDB, 5 (8) : 776-787, 2012年。

[3]史蒂夫●贝斯特。JFS概述。http://jfs.sourceforge.净/项目/发布/jfs.pdf, 2000.

[4] John S. Bucy, Jiri Schindler, Steven W. Schlosser和Gregory R. Ganger。DiskSim Simulation Environment 4.0版参考手册。卡内基梅隆大学技术报告CMU-PDL-08-101, 2008年5月。

[5] Remy Card, Theodore Ts'o和Stephen Tweedie。第二个扩展文件系统的设计和实现。在1994年12月于荷兰阿姆斯特丹举行的第一届Linux国际荷兰国际研讨会上。

[6] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau和Remzi H. Arpaci-Dusseau。一致性，无需订购。在FAST '12中，第101–116页，加利福尼亚州圣何塞，2012年2月。

[7] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules和Alistair Veitch。LazyBase：在可扩展的数据库中以新鲜度换取性能。在EuroSys '12，第169-182页，瑞士伯尔尼，2012年4月。

[8] Jonathan Corbet。障碍和日记文件系统。http://LWN.NET/TrimeSe/83161, 2008年5月。

[9] Jonathan Corbet。那庞大的文件系统线程。http://LWN.NET/TUNESS/32641/1/, 2009年3月。

[10] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka和Lei Zhang。通用文件系统依赖性。在SOSP '07，第307-320页，华盛顿州史蒂文森，2007年10月。

[11] Gregory R. Ganger和Yale N. Patt。文件系统中的元数据更新性能。在OSDI '94，第49-60页，加利福尼亚州蒙特雷，1994年11月。

[12]莫里斯●赫利希（Maurice Herlihy）。道歉与要求许可：抽象数据类型的乐观并发控制。ACM Transactions on Database Systems (TODS)，15 (1) : 96–124，1990年。

[13] DM Jacobson和J. Wilkes。基于旋转位置的磁盘调度算法。技术报告HPL-CSP-91-7，惠普实验室，1991年。

[14] KnowledgeTek。 串行ATA规范修订版3.0金。http://www.knowledgetek.com/datastorage/ 培训课程/SATA_3.0-8.14.09(CD).pdf, 2009.

[15] Charles M. Kozierok。SCSI接口的概述和历史记录。http://www.pcguide.com/ref/hdd/if/scsi/ 超C语言,2001.

[16]祥宗和约翰●鲁滨逊。关于并发控制的乐观方法。ACM Transactions on Database Systems (TODS)，6 (2) : 213–226，1981年。

[17]理查德●麦克杜格尔和吉姆●毛罗。Filebench。http://SureFurg.NET/Apps/MediaWiKi/FielBistor index.php?title =文件平台, 2005.

[18]马歇尔●柯克●麦克库西克。从文件系统角度看磁盘。ACM通讯，55 (11)，2012年。

[19] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen和Jason Flinn。重新考虑同步。在OSDI '06，第1-16页，华盛顿州西雅图，2006年11月。

[20] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu 和 Erez Zadok。I^{3}FS: 内核完整性检查程序和入侵检测文件系统。在 LISA '04，第 69-79 页，乔治亚州亚特兰大，2004 年 11 月。

[21] Zachary NJ Peterson。使用虚拟连续性进行写时复制的数据放置。硕士论文，加州大学圣克鲁斯分校，2002 年。

[22] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。日记文件系统的分析和发展。在 USENIX '05，第 105-120 页，加利福尼亚州阿纳海姆，2005 年 4 月。

[23] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。铁文件系统。在 2005 年 10 月，英国布莱顿，SOSP '05，第 206-220 页。

[24] Margo Seltzer, Peter Chen 和 John Ousterhout。再谈磁盘调度。1990 年 1 月，在 USENIX 90 冬季刊，第 313-324 页，华盛顿特区。

[25] Margo I. Seltzer, Gregory R. Ganger, M。Kirk McKusick, Keith A. Smith, Craig AN Soules 和 Christopher A. Stein。日记与软更新：文件系统中的异步元数据保护。在 USENIX '00，第 71-84 页，加利福尼亚州圣地亚哥，2000 年 6 月。

[26]迪克站点。我的磁盘有多快？威斯康星大学麦迪逊分校系统研讨会，2013 年 1 月。http://www. C.WISC EDU/ECTION/HO-FAST MY 磁盘.

[27]大卫●所罗门。在 Windows NT 中。Microsoft 编程系列。Microsoft 出版社，第二版，1998 年 5 月。

[28]乔恩●索尔沃思和西里尔●奥吉。只写磁盘缓存。在

SIGMOD '90，第 123-132 页，新泽西州大西洋城，1990 年 5 月。

[29] SQLite 小组。如何损坏一个 SQLite 数据库文件。http://www.sqlite.org/howtocorrupt.html，2011.

[30] Marting Steigerwald。强制命令。Linux 杂志，2007 年 5 月。

[31] Christopher A. Stein, John H. Howard 和 Margo I. Seltzer。统一文件系统保护。在 USENIX '01，第 79-90 页，2001 年 6 月，马萨诸塞州波士顿。

[32] Adan Sweeney, Doug Doucette, Hui Hu, Curtis Anderson, Mike Nishimoto 和 Geoff Peck。XFS 文件系统中的可伸缩性。在 1996 年 1 月于加利福尼亚州圣地亚哥的 USENIX 1996 中。

[33]左宗棠。回复：[PATCH 0/4]（重新发送）ext3 [34]屏障更改。Linux 内核邮件列表。http://articl e. GMANE.Org/GMANE.COF.FielSysExt/Ext4/ 6662，2008 年 5 月。

[34] Theodore Ts'o 和 Stephen Tweedie。Ext2 / 3 文件系统的未来方向。2002 年 6 月，在加利福尼亚州蒙特雷的 FREENIX '02 上发表。

[35] Stephen C. Tweedie。记录 Linux ext2fs 文件系统。1998 年 5 月，在北卡罗来纳州达勒姆市举行的第四届年度 Linux 博览会上。

[36] Yang Wang, Manos Kapritsos, Zuocheng Ren, Mahajan 王子, Jeevitha Kirubanandam, Lorenzo Alvisi 和 Mike Dahlin。Salus 可扩展块存储中的稳健性。在 NSDI '13，第 357-370 页，伊利诺伊州伦巴底，2013 年 4 月。

[37]拉尔夫●韦伯。SCSI 体系结构模型-3（SAM-3）。http://www.t10.org/ftp/t10/drafts/sam3/

SAM3R14PDF，2004 年 9 月。

[38] BL Worthington, GR Ganger 和 YN Patt。现代磁盘驱动器的调度算法。在 SIGMETRICS '94，第 241-251 页，田纳西州纳什维尔，1994 年 5 月。

[39]于海峰和阿明●瓦达特。复制服务的连续一致性模型的设计和评估。在 OSDI '00，加利福尼亚州圣地亚哥，2000 年 10 月。